Final Report

A Single Chip VLSI Implementation of a QPSK/SQPSK Demodulator For A VSAT Receiver Station.

Submitted to:

NASA Lewis Research Center
21000 Brookpark Road
Cleveland, OH 44135

Submitted by:

Dr. S.C. Kwatra
Principal Investigator

Brent King
Graduate Research Assistant

Department of Electrical Engineering
College of Engineering
University of Toledo
Toledo, OH 43606

Report No. DTVI - 43

June 1995

106080
248 p.

Final Report

A Single Chip VLSI Implementation of a QPSK/SQPSK Demodulator For A VSAT Receiver Station.

Submitted to:

NASA Lewis Research Center
21000 Brookpark Road
Cleveland, OH 44135

Submitted by:

Dr. S.C. Kwatra
Principal Investigator

Brent King
Graduate Research Assistant

Department of Electrical Engineering
College of Engineering
University of Toledo
Toledo, OH 43606

Report No. DTVI - 43

June 1995

An Abstract of

A Single Chip VLSI Implementation of a QPSK/SQPSK
Demodulator For A VSAT Receiver Station.

By Brent King

Submitted as partial fulfillment of the requirements for
the Master of Science Degree in
Electrical Engineering.

University of Toledo
June 1995

This thesis presents a VLSI implementation of a QPSK/SQPSK demodulator. It is designed to be employed in a VSAT earth station that utilizes the FDMA/TDM link. A single chip architecture is used to enable this chip to be easily employed in the VSAT system. This demodulator contains lowpass filters, integrate and dump units, unique word detectors, a timing recovery unit, a phase recovery unit and a down conversion unit.

The design stages start with a functional representation of the system by using the C programming language. Then it progresses into a register based representation using the VHDL language. The layout components are designed based on these VHDL models and simulated. Component generators are developed for the adder, multiplier, read-only memory and serial access memory in order to shorten the design time. These sub-components are then block routed to form the main components of the system. The main components are block routed to form the final demodulator.

Acknowledgments

I would like to express my deep and sincere appreciation to Dr. S.C. Kwatra for his time, patience and support. I would also like to thank Dr. A.G. Eldin for his countless answers to my countless questions. It is a privilege to work with such knowledgable professors. Their guidance has been invaluable to my research. I would like to thank Robert E. Jones and Dr. Ed Smith for their participation in the project and for serving as members of my committee.

My thanks also go out to William B. Curry II who helped me endlessly with the communications concepts involved in my research. Also Dinraj Shetty, Subhash Chintamaneni, and Mohan Pakurti helped me in learning and implementing the VLSI tools.

Lastly, I would like to express my thanks to my Mother. She has been a real inspiration and has supported me from the very beginning. I know that she will be the happiest person on the day of my graduation.

This report contains part of the work performed under NASA grant NAG3-799 during the period January 1994 to June 1995. The research was performed as part of the Master's thesis requirement for Mr. Brent King.

Dr. S.C. Kwatra

Principal Investigator

Table of Contents

List of Figures

Chapter 1 Introduction 1

Chapter 2 System Background 10

Chapter 3 Architecture 18

Chapter 4 Generators 38

List of Figures

List of Figures

List of Figures

Chapter 8 Conclusions 113

List of Tables

Chapter 1 Introduction

1.0 Recent Advances

Recent advances in technology have prompted a new era in communication systems. Smaller, faster chips as well as more powerful and efficient antennas are just a few of these advances. This is encouraging the renovation of existing systems so that they will become higher in performance as well as conform to the needs of more users. The satellite industry is now applying these advances to the satellite communication systems and are creating systems that are smaller, less complex and more user oriented. The systems that were once dominated by the government and large industry will now be shared by smaller organizations.

1.1 VSAT Terminals

This renovation is pointing to a new generation of earth stations which are known as very small aperture terminals (VSAT) [1]. The antenna apertures for VSAT stations are to be approximately one meter in diameter, greatly reducing the size of the system.

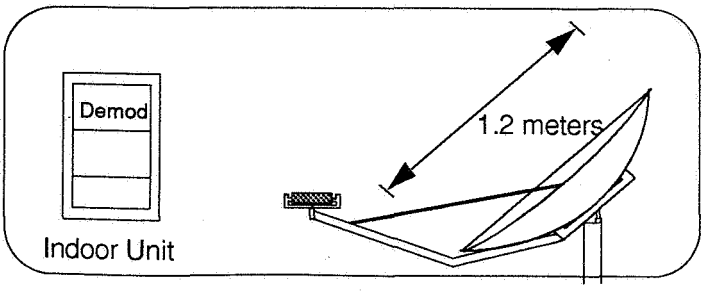


Fig. 1.0 VSAT Antenna

VSAT will have advantages over existing terrestrial systems such as a lower operating cost, ease of installation and maintenance, and support for multiservices. Some VSAT applications will be credit authorization, long distance voice and data communications, and electronic mail. A network of VSAT terminals will be placed in a star formation and will communicate to a central hub station via satellite. The signals will be transmitted and received in a statistically bursty fashion and the users are expected to share the satellites resources in a cooperative manner.

1.2 Demodulator

Earth stations are composed of many complex components. Reducing the number and complexity of these components will improve the performance of the entire system. One way to do this is to design a better earth station demodulator.

1.3 Previous Work

Previous research is done by Dave Wagner at the University of Toledo [2]. He accomplished an architectural design as well as a layout design of an earthstation demodulator but was unable to meet some of the targeted performance requirements of the design. The leading factors in the degradation of the performance are the CMOS technology used and the tools that were available. The 1.2μ CMOS technology is used in his design which is quite large compared to the 0.8μ technology used today. It is very difficult to fit such a large design in a frame area of $1cm^2$ using this CMOS process. The layout tool used in Dave's research is called Magic, which is an introductory layout editor and is not

up to industry standards. It is very difficult to design large circuits with Magic, so public circuit libraries from other universities are used to gather the larger circuits, such as the read-only memory (ROM). This resulted in circuits that are not optimized to work together in the system and, therefore, degraded the performance of the demodulator.

1.4 Objective of This Work

The objective of this research is to design a high performance QPSK/SQPSK demodulator that can be used in an FDMA/TDM VSAT system. In an effort to implement all analog circuitry with digital circuitry, this demodulator will be equipped with a digital downconversion unit. The only requirement is that the transmitted signal be passed through an A/D converter. All of the components of the demodulator will be placed onto a single chip, increasing the performance as well as decreasing the cost and size of the system. Throughout this research, the emphasis will be on decreasing the complexity of the system, which includes designing a demodulator with minimal external control signals. The process technology that will be used to design the demodulator is 0.8μ CMOS technology developed by Hewlett Packard.

1.4.1 System Specifications

Initial system specifications will be used in order to create a starting point in the design stage. The specifications that will be used are listed below in Table 1:

Table 1: System Specifications

Demodulation	QPSK/SQPSK
Frame Length	30 ms
IF Input	25 MHz
Data Rate	50 Mbps

The frame length is the total amount of time that it takes for the satellite to communicate with all of it's assigned earth stations. The assumption made is that there are 10 earth stations that the satellite will communicate with in 3ms bursts. Fig. 1.1 shows a diagram of a typical frame schedule, where the satellite communicates with earth station A through earth station J and then repeats.

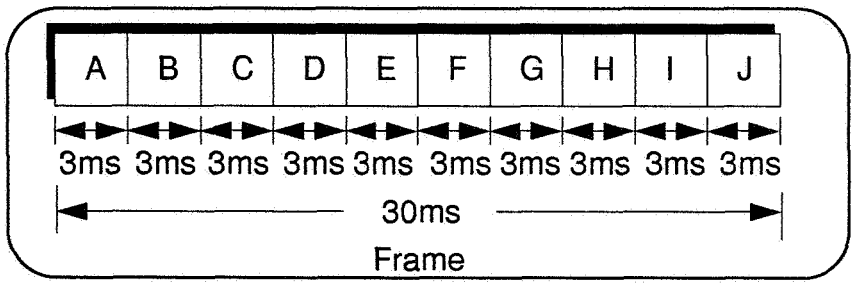


Fig. 1.1 Frame Length

Efficient signal transmission is done at very high radio frequencies (RF). Efficient processing of the signal should be done at lower intermediate frequencies (IF) because today's technology does not allow processing at RF very easily. Therefore, the RF signal must be translated into an equivalent IF signal before it enters the demodulator. To find the proper IF range, the data rate must be inspected first. High performance demodulators require very high data rates. The target data rate for this demodulator is chosen to be 50MHz. The output data is obtained by multiplexing the symbols in the I and Q channels so the symbol rate

will be half of the data rate as shown in Fig. 1.2. This makes the symbol rate equal to 25MHz.

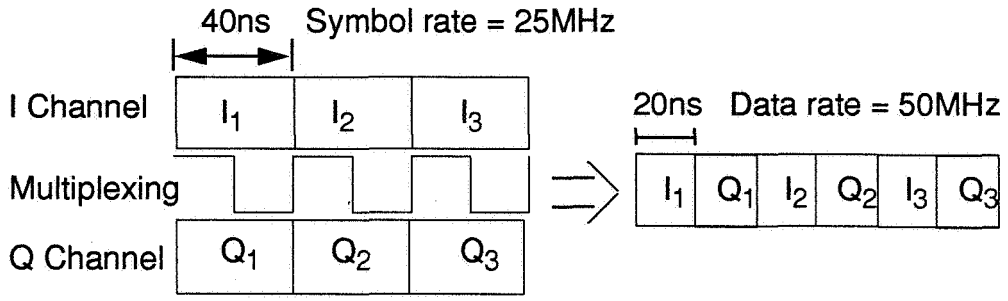


Fig. 1.2 Symbol Rate and Data Rate

Each symbol will be sampled 4 times which leads to a sampling rate of 100MHz. The Nyquist theorem states that a signal should be sampled at a rate at least twice it's highest frequency. If the IF is set to 25MHz, then the Nyquist theorem is satisfied. Fig. 1.3 shows the power spectrum of the message signal M(f) and the transmitted signal S(f) with an IF of 25MHz. The transmitted signal's highest frequency component is 50MHz so the sampling rate can be at least 100MHz. A sampling rate of 100MHz corresponds to a 10ns period in which the samples will need to be processed by the components in the demodulator..

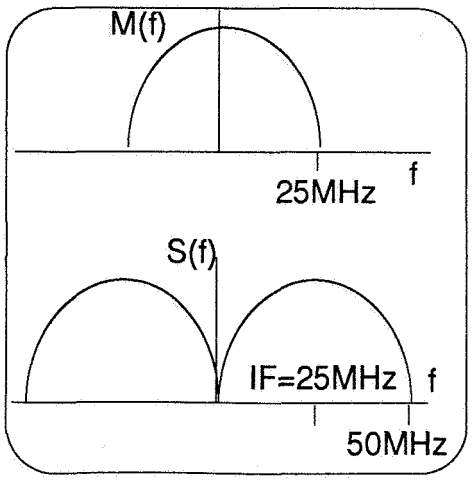


Fig. 1.3 Signal Spectrums

1.4.2 Tools Used

The tools available for this research are much more sophisticated than the earlier tools. The University has purchased CAD tools owned by Mentor Graphics which give the students industrial experience in VLSI design work. Mentor Graphics tools are user friendly and have a wide variety of user interfaces to help in the design stages. The Mentor Graphics software tools used in this research are VHDL, VHDLsim, Led, Lx, Lsim and MicroRoute.

VHDL is used to describe hardware for the purpose of simulating, modeling, testing, and designing digital systems. A behavioral model written in the VHDL language describes the operation as well as the delay of a component. Many VHDL components can be connected together to form a circuit structure which can then be simulated. Mentor Graphics Explorer VHDLsim is an electronic design tool that simulates the behavior of digital circuits that have been written in the VHDL language. Some advantages of using VHDL are the ease of interchanging components during the design stage and the short simulation time needed.

The Mentor Graphics Led is used to design the circuitry that will be used in the digital system. Led supports two levels of transistor representation: schematic capture level and layout level representation. The schematic capture level representation allows the designer to quickly assemble circuits and obtain some initial transistor sizes. The layout representation allows the designer to place the transistors exactly as they will be when fabricated. The different metal layers are also specified in this representation. The layout representation possesses all of the characteristics that are needed to fabricate the circuit, which includes all transistor sizes as well as final circuit area.

Mentor Graphics Lsim is a multi-level simulator. There are three different modes of simulation that cater to the needs of the designer. These different modes are the switched mode, adept mode and mixed mode operation. When simulating large circuits or just checking the connectivity of a circuit, the switched mode simulation allows for very fast results. Each transistor is represented as a switch and does not have the characteristics of a true transistor, so the simulation results have no delay information nor does it have the characteristics of the circuit's true output. The adept mode is a realistic representation of simulation where each transistor is described by an equation. This calculation intensive procedure takes a vast amount of time but it results in a reliable output waveform. The mixed mode operation allows both switched simulation and adept simulation to occur at the same time.

Lx is a procedural interface to the L database and Led graphics editor. It is built from a set of database interface functions and from general purpose language called GENIE. Lx provides access to the information within the L database and provides interaction with the Led graphics editor. This interface is an important tool in the development of component generators.

The circuit routing is accomplished using Mentor Graphics MicroRoute. An initial component placement and a netlist that describes the connectivity among the components is all that is needed. It then creates channels for which it will route the components. Pre-routing is done for power and ground lines in order to keep them regular. Finally, routing is completed channel by channel until the entire circuit is routed.

1.5 Chapter Summaries

Chapter 2 contains background information pertaining to the

proposed VSAT system. It covers the basics behind QPSK and SQPSK modulation and some of the advantages of each of these modulation schemes. The FDMA/TDM satellite link is illustrated, and the preamble structure for this type of link is covered.

The architectures of all of the main components are introduced in Chapter 3. These architectures are chosen such that they are easily implemented with digital circuitry as well as quickly assembled. Other architectural decisions are based on the performance of the algorithm.

Chapter 4 shows the design stages for the commonly used sub-components that will be used in the system. Generators are developed for the binary adder, multiplier, read-only memory and serial access memory.

The design of the main blocks is covered in Chapter 5. The main blocks are composed of the generated components and glue logic. MicroRoute is used to do the block routing of these sub-components to form the main components.

Chapter 6 illustrates the organization of the chip. A tree buffering scheme is used in order to equalize the delay to all of the components.

Simulation and verification are introduced in Chapter 7. Behavioral level and register level simulations are done on the system. Test vectors are used to verify the connectivity and the functionality of all of the component blocks.

The conclusions and future research are found in Chapter 8. Various architectural improvements are proposed for future research.

The appendices contain the various code that are used throughout the research. Appendix A has the Lx code used for the various generators that are developed for the research. Appendix B contains some MicroRoute tips that will be beneficial to any designer who uses this router. All of the VHDL code used to represent the different components is in Appendix C. The functional system

representation implemented with C code, as well as the code used to program the ROMs, is in Appendix D.

Chapter 2 System Background

2.0 Quadrature Phase Shift Keying

The digital modulation technique chosen should provide a reliable performance, a low probability of error and an efficient utilization of the channel bandwidth. Quadrature phase shift keying (QPSK) has these characteristics which is why it is one of the most popular of the digital modulation schemes [3]. The inphase and quadrature components (b_i, b_q) of the QPSK modulated carrier are given as:

$$b_i = \sqrt{\frac{2E}{T}}\left(\cos\frac{2\pi i}{4}\right) \quad \text{and} \tag{2.1}$$

$$b_q = \sqrt{\frac{2E}{T}}\left(\sin\frac{2\pi i}{4}\right) \tag{2.2}$$

where E is the symbol energy, T is the symbol duration and i=0,1,2 or 3. These bit streams make step changes at the same time to create a transmitted wave that has four possible phases: 0, $\pi/2$, π, and $3\pi/2$. This results in four distinct symbols, 00, 01, 10, and 11, which are represented by the constant envelope carrier:

$$s_i(t) = \sqrt{\frac{2E}{T}}\cos\left(\frac{2\pi i}{4}\right)\cos(2\pi f_c t) - \sqrt{\frac{2E}{T}}\sin\left(\frac{2\pi i}{4}\right)\sin(2\pi f_c t) \tag{2.3}$$

where f_c is the center frequency. At a symbol transition, the QPSK wave can go

through a 0, ±90 or ±180 degree phase change, depending on what the two consecutive symbols are. A symbol change from 00 to 01 corresponds to a 90 degree change because only a single bit has changed, while a transition from 00 to 11 will cause a 180 degree change. A simplified model of a modulator is shown in Fig. 2.0.

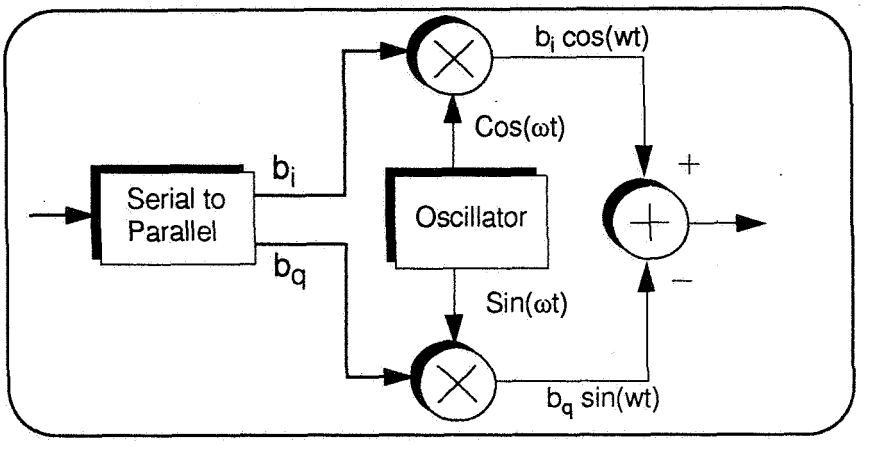


Fig. 2.0 QPSK modulator

2.1 Staggered Quadrature Phase Shift Keying

Staggered quadrature phase shift keying (SQPSK), also known as offset QPSK (OQPSK), is a modification of the QPSK modulation scheme. An SQPSK modulator is shown in Fig. 2.1. The two bit streams b_i and b_q are staggered by one bit duration.

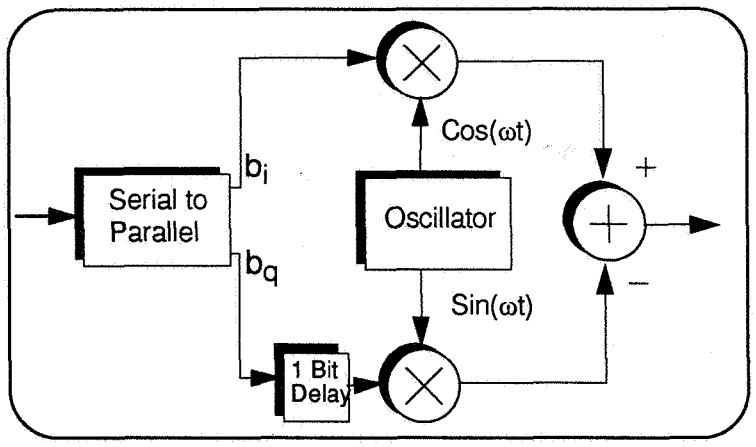


Fig. 2.1 SQPSK Modulator

This insures that only a single bit changes at a time during a symbol duration causing the SQPSK wave to go through a 0 or a ±90 degree phase change. Because orthogonality is still preserved through the symbol duration, the power spectral density and the average probability of error are the same for both QPSK and SQPSK. This is always true unless there are nonlinearities in the channel.

QPSK and SQPSK signals are both characterized by a constant envelope. This is important because of the amplitude nonlinearities that are apparent in a satellite channel. When these signals are passed through a filter or a band limiter, the envelope no longer stays constant. During a 90 degree phase change, the envelope's amplitude will only change by 3dB. A 180 degree phase change will cause the envelope to go through a zero transition. Now the amplitude nonlinearities can seriously degrade this signal. An advantage that SQPSK has over QPSK is that it's envelope will only go through a 3dB change at the most since the maximum phase change that it can go through is 90 degrees. This is shown in Fig. 2.2.

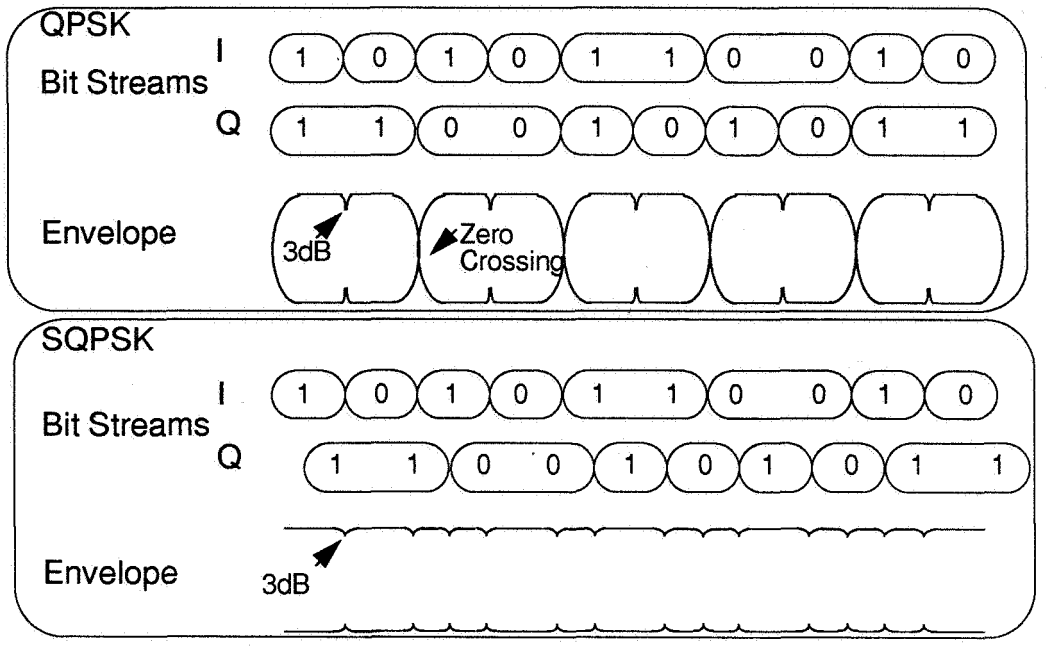


Fig. 2.2 Band Limited QPSK and SQPSK

2.2 Gray Coding

Additive white gautian noise (AWGN) is unwanted power that is added to the transmitted signal by the channel. AWGN may add just enough power to cause the transmitted symbol to be incorrectly regenerated. This will most likely cause a symbol to be mistaken for one of it's adjacent symbols as illustrated in Fig. 2.3.

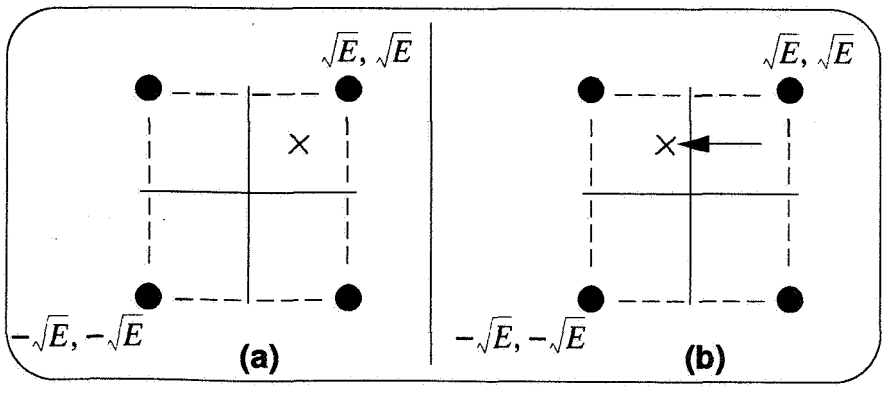


Fig. 2.3 a) Transmitted Symbol, b) Effect of Noise on Symbol

For this reason, Gray coding is a very important consideration when doing modulation. It places the symbols such that adjacent phases only differ by 1 bit. Now if AWGN causes this type of error, there will only be at most one bit error.

2.3 Demodulation

A basic block diagram of a QPSK demodulator is shown in Fig. 2.4. The modulated signal is downconverted from IF to baseband by multiplying it by Sin(ωt) and Cos(ωt). This also splits the inphase and quadrature components into two channels (I,Q). The lowpass filter eliminates the unwanted frequency components and noise that reside outside the desired signal's bandwidth. The I and Q channel signals are integrated over one symbol period and a decision is made as to what logic level the symbol is. For demodulation and detection, phase recovery and timing recovery are needed. The phase recovery unit uses a nonlinear transform and an averaging technique to estimate the phase of the incoming carrier. This is important in QPSK/SQPSK demodulation because the information is carried on the phase of the modulated wave. The timing recovery unit extracts the symbol clock from the carrier wave, keeping the receiver synchronized with the transmitter.

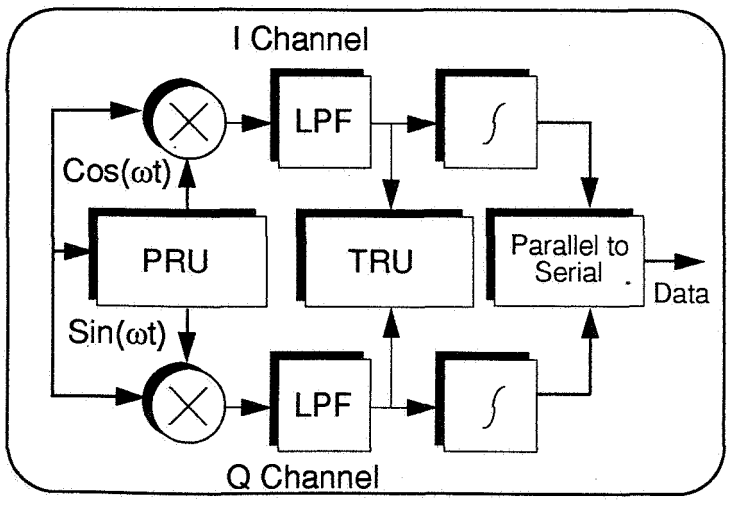


Fig. 2.4 Block Diagram of Demodulator

2.4 FDMA/TDM Link

The satellite link must be governed by an efficient access scheme in order to adhere to high quality standards. A typical link setup for the QPSK satellite network is the FDMA/TDM link.

Frequency division multiple access (FDMA) is used in the uplink and allows multiple earth terminals to communicate with a single satellite simultaneously. This is done by dividing the frequency spectrum into multiple bands and allotting each earth station its own frequency band for which to transmit. Fig. 2.5 shows four earth terminals transmitting signals in their respective bands.

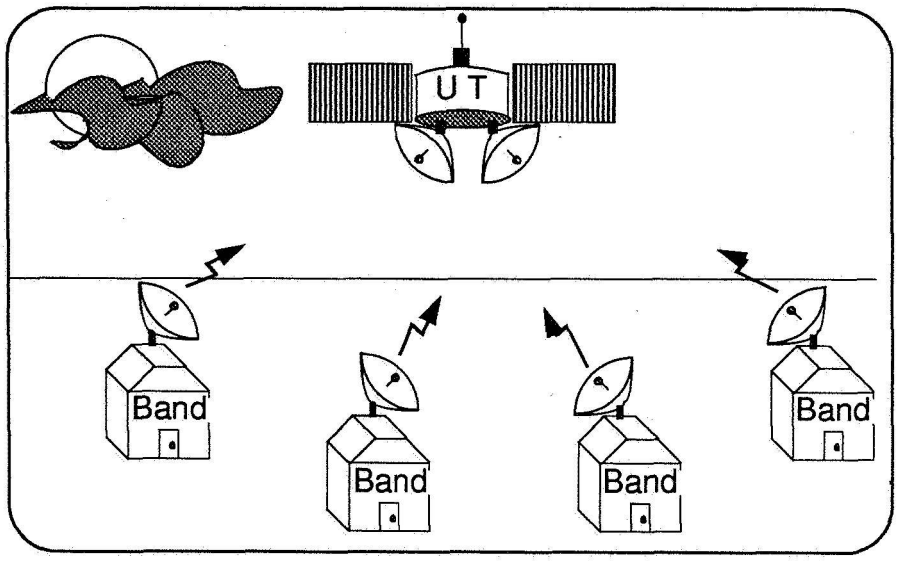


Fig. 2.5 FDMA access scheme.

Time division multiplexing (TDM) is a downlink access scheme that increases the performance of the system and is very easy to implement with digital modulation. A single satellite will communicate with several earth terminals

in different time slots as shown in Fig. 2.6.

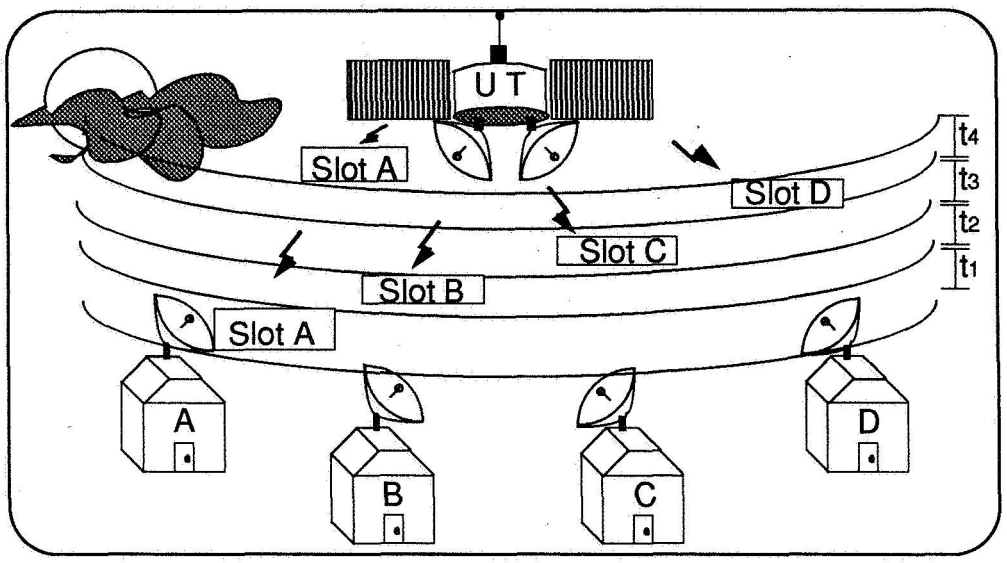


Fig. 2.6 Time division multiplexing scheme.

The satellite will communicate to the earth terminals in short bursts in order to avoid keeping any of the earth stations waiting for long periods of time. The groups of signals will take turns using the channel, allowing the earth terminal to utilize the entire bandwidth and power resources of the satellite.

2.5 Slot Format

Due to the bursty nature of TDM, there is a need to provide preamble bits to recover the phase and timing of the transmitted signal. The preamble is a group of extra bits that will allow the receiver to quickly synchronize itself with the transmitter. It is composed of carrier recovery (CR), timing recovery (TR) and unique word (UW) bits that work with various units in the demodulator to recover the signal correctly. A phase recovery unit will utilize the CR bits in order to create an initial phase lock. A timing recovery unit uses the TR bits to create an initial timing of the signal. The unique word detector uses the UW bits to tell when the data starts and also to resolve a phase ambiguity caused by the phase

recovery unit. Later chapters will go into more details about these signals. The slot format is shown in Fig. 2.7.

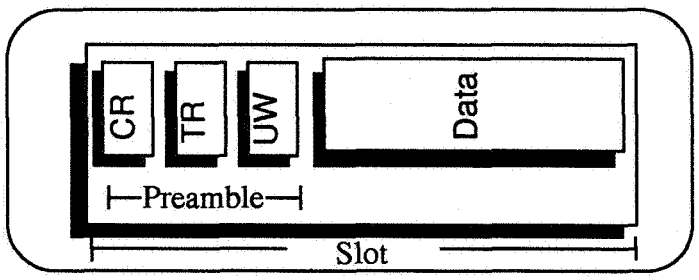


Fig. 2.7 Slot Format.

Chapter 3 Architecture

3.0 The Demodulator

The demodulator is made up of many components which work together to extract the information from the transmitted signal. A block diagram of the demodulator is shown in Fig. 3.0.

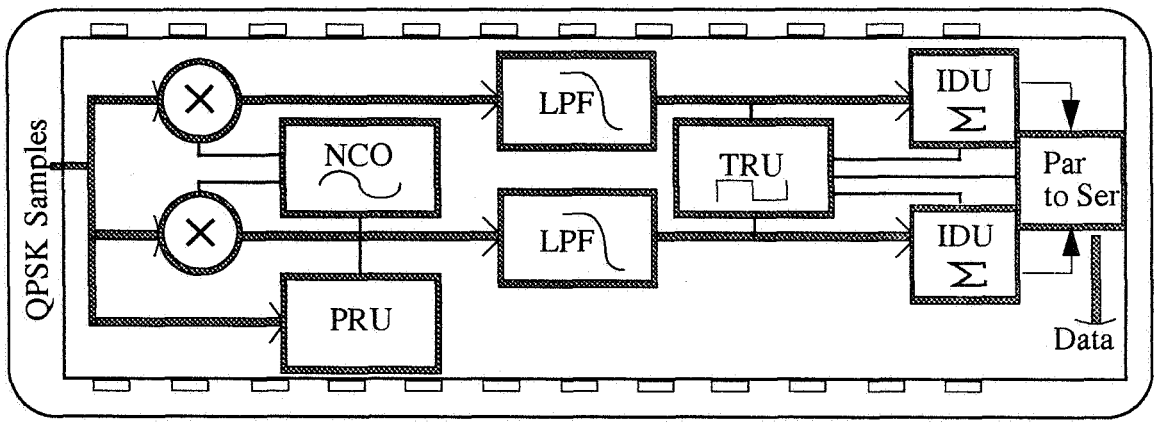


Fig. 3.0 Demodulator Block Diagram

An analog to digital converter (A/D) will sample the transmitted signal and feed the samples to the input of the demodulator. From there, the signal is fed into a correlation receiver which downconverts the signal, lowpass filters it and makes a decision as to which bit is sent. Care is needed in designing the components because the performance of the demodulator is based on the performance of it's components. The architectures of all of the components are explained in detail in the sections that follow.

3.1 Correlation Receiver

The correlation receiver is shown in Fig. 3.1.

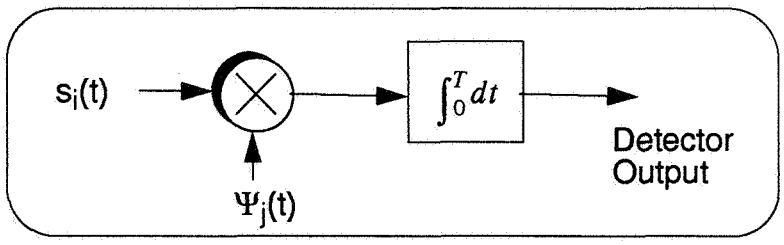


Fig. 3.1 Correlation Receiver

It correlates the received signal $s_i(t)$ with a known replica $\Psi_j(t)$ and compares it's output, after one symbol period, to a threshold value to make a decision as to what bit is transmitted.

Assume that the input signal $s_i(t)$ is being passed through a linear filter with an impulse response of $h_j(t)$. The output of the filter will be:

$$y(t) = \int_{-\infty}^{\infty} s_i(\tau)\, h_j(t-\tau)\, d\tau \tag{3.0}$$

If the linear response is set to

$$h_j(t) = \Psi_j(T-t) \tag{3.1}$$

Then the resulting filter output will be

$$y(t) = \int_{-\infty}^{\infty} s_i(\tau)\, \Psi_j(T-t-\tau)\, d\tau \tag{3.2}$$

If the output is then measured at time t = T, the output becomes

$$y(T) = \int_{-\infty}^{\infty} s_i(\tau)\, \Psi_j(\tau)\, d\tau \tag{3.3}$$

Since $\Psi_j(t)$ is defined to be zero outside the interval $0 \le t \le T$, the output of the filter becomes

$$y(T) = \int_{-\infty}^{\infty} s_i(\tau)\, \Psi_j(\tau)\, d\tau \tag{3.4}$$

which represents exactly what Fig. 3.1 will perform. Therefore, the filter whose

impulse response is a time-reversed and delayed version of the signal $s_i(t)$ is said to be matched to the signal $s_i(t)$ [3], and since $s_i(t)$ is a replica of $\Psi_j(t)$, the time-reversed and delayed version of the signal $\Psi_j(t)$ is said to be matched to the signal $s_i(t)$.

This matched receiver is optimum for the detection of a pulse in additive white gaussian noise (AWGN). However, the timing recovery unit used in this demodulator requires that the baseband signal be reconstructed. Therefore, a lowpass filter is placed in the correlation receiver, as shown in Fig. 3.2, causing the matched filter to become sub-optimum.

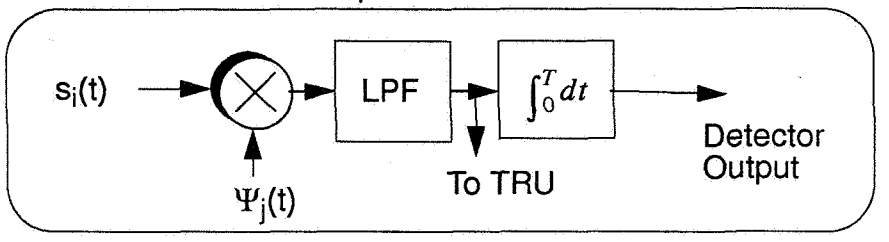


Fig. 3.2 A QPSK/SQPSK Demodulator

A QPSK/SQPSK demodulation requires that there be two correlation receivers as shown in Fig. 3.3.

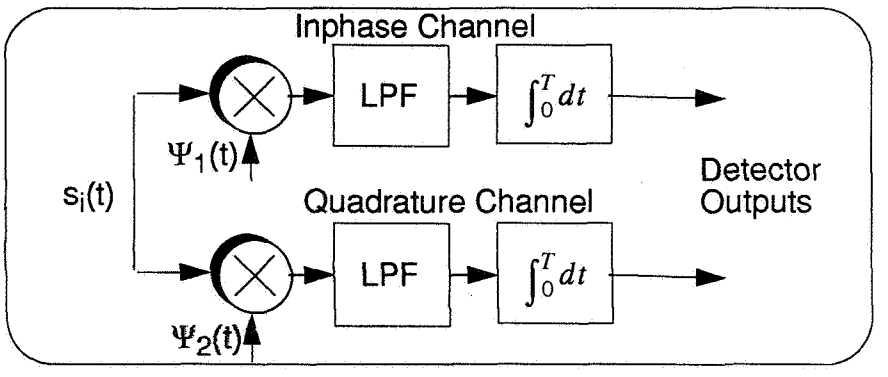


Fig. 3.3 QPSK/SQPSK Correlation Receiver

This results in two different channels called the inphase and quadrature channels. A numerically controlled oscillator is used to provide the $\Psi_1(t)$ and $\Psi_2(t)$ signals, the Parks/McClellan lowpass filter is used to reconstruct the baseband signal and

an integrate and dump unit is used to do the integration over one symbol period and make the bit decision. These components will be described in the following sections.

3.1.1 Numerically Controlled Oscillator

Analog oscillators, in a QPSK demodulator, output the signals $\cos(2\pi f_c t+\theta_e)$ and $\sin(2\pi f_c t+\theta_e)$ to downconvert the modulated signal, where θe is a phase estimate from a phase recovery unit. The modulated signal is then separated into the inphase and quadrature channels. In an effort to replace analog circuitry with digital circuitry, a numerically controlled oscillator (NCO) has been developed. The NCO will produce samples of $\sin(2\pi f_c nT+\theta_e)$ and $\cos(2\pi f_c nT+\theta_e)$ at each sampling instance. A phase increment of $2\pi f_c T$, where T is the sample period, will be accumulated, as shown by the NCO in Fig. 3.4, in order to create the argument $2\pi f_c nT$ where n=0,1,... is the sample time. The phase estimate from the PRU will also be added to form the argument $2\pi f_c nT + \theta_e$. The inphase and quadrature samples are given as:

$$I(nT) = A\cos\left(2\pi f_c nT + \frac{2\pi i}{4} + \theta_{in}\right)\cos\left(2\pi f_c nT + \theta_e\right) \tag{3.5}$$

$$Q(nT) = A\cos\left(2\pi f_c nT + \frac{2\pi i}{4} + \theta_{in}\right)\sin\left(2\pi f_c nT + \theta_e\right) \tag{3.6}$$

where θ_{in} is the initial phase of the carrier. The sin(•) and cos(•) outputs of these arguments can be stored in a ROM, which is used as a look-up table.

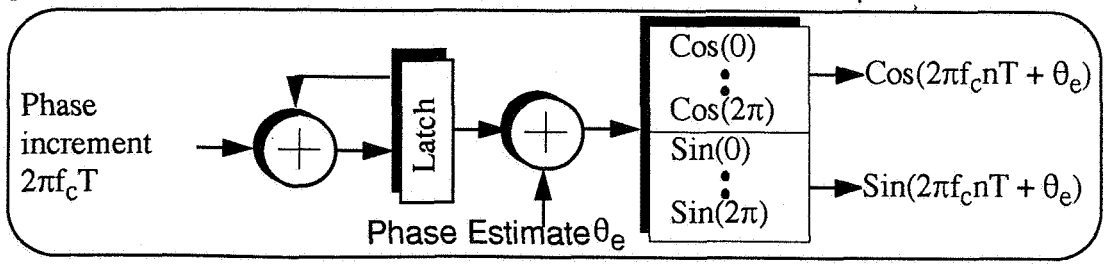


Fig. 3.4 Numerically Controlled Oscillator

3.1.2 Multiplier

Multipliers are used in many digital signal processing applications. Many algorithms have been formalized to do multiplication. The Baugh/Wooley algorithm [5] is used in high speed, two's complement multiplication. The main advantage of this algorithm is that the signs of all the partial product bits are positive, allowing the product to be formed using array addition techniques. A modified version of the Baugh/Wooley algorithm has been developed in [6]. It saves three adder cells over the original algorithm. A block diagram of this multiplier is shown in Fig. 3.5.

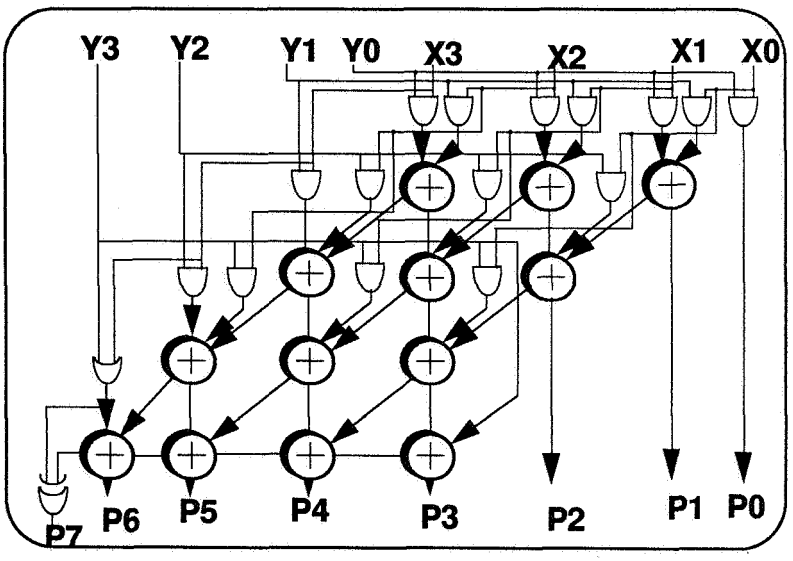


Fig. 3.5 Multiplier Architecture

3.1.3 Lowpass Filter

The digital lowpass filter (LPF) is used to reconstruct the baseband signal after the downconversion is done. The inputs to the LPF are given in equations (3.0) and (3.1). Using a well known trigonometric identity, these equations can be re-written as:

$$I(nT) = A'\cos\left(\frac{2\pi i}{4} + \theta_{in} - \theta_e\right) + A'\cos\left(2\pi 2 f_c nT + \frac{2\pi i}{4} + \theta_{in} + \theta_e\right) \qquad (3.7)$$

$$Q(nT) = A'\sin(\frac{2\pi i}{4} + \theta_{in} - \theta_e) + A'\sin(2\pi 2f_c nT + \frac{2\pi i}{4} + \theta_{in} + \theta_e) \quad (3.8)$$

The LPFs are designed to pass all frequency components from 0 to 25MHz. Therefore, the filters will eliminate the second term in both (3.2) and (3.3) since thesis components are centered at $2f_c$ = 50MHz. After the filters, the inphase and quadrature signals will be:

$$I(nT) = A'\cos(\frac{2\pi i}{4} + \theta_{in} - \theta_e) \quad (3.9)$$

$$Q(nT) = A'\sin(\frac{2\pi i}{4} + \theta_{in} - \theta_e) \quad (3.10)$$

If the initial phase of the carrier is estimated (θ_i=θ_e) by the phase recovery unit, then the signals become

$$I(nT) = A'\cos(\frac{2\pi i}{4}) \quad (3.11)$$

$$Q(nT) = A'\sin(\frac{2\pi i}{4}) \quad (3.12)$$

which are replicas of the transmitted bit sequences shown in (2.1) and (2.2).

A finite impulse response (FIR) filter is a digital filter whose response to a unit impulse is finite in duration. FIR filters are characterized by their constant delay and stability. A typical amplitude response of an FIR approximation to an ideal lowpass filter is shown in Fig. 3.6

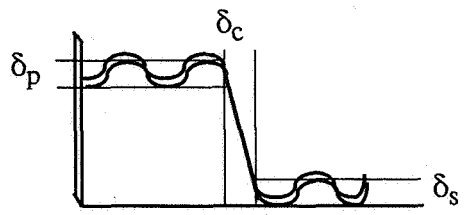


Fig. 3.6 Typical FIR Approximation of an Ideal LPF

It is advantageous to minimize δp, δc and δs in order to get the characteristics of an ideal LPF, where δp is the amount of ripple in the passband, δc is the transition distance at cutoff, and δs is the amount of ripple in the stopband. The Chebyshev approximation is one way to do this. It's main goal is to minimize the maximum error of the output of the filter. A Parks and McClellan algorithm uses the Chebyshev approach to generated the filter coefficients.

Digital filtering for FIR filters can be described by the equation:

$$y(n)) = \sum_{k=0}^{N-1} h(k)\,x(n-k) \qquad (3.13)$$

where h(k) are the coefficients of the filter, k=0,1,..., N-1, and x(n) and y(n) are respectively the input and output of the filter. The values of the filter coefficients determine the filter's characteristics. Equation (3.13) shows that filtering is simply the convolution of the input signal with the filter's impulse response in the time domain. This filtering operation can also be shown in the z-domain which is analogous to the frequency domain. Convolution in the time domain is equal to multiplication in the frequency domain, so the filtering operation can be done as follows:

$$Y(z) = H(z)\,X(z) \qquad (3.14)$$

where H(z) is the z-transform of the filter's impulse resp0onse, and X(z) and Y(z) are the z-transforms of the input sequence and output sequence of the filter.

The transfer function for the Parks/McClellan FIR filter is [2]:

$$H(z) = 0.5 + 0.316(z^{1}+z^{-1}) - 0.1(z^{3}+z^{-3}) + 0.055(z^{5}+z^{-5}) - 0.034(z^{7}+z^{-7}) \quad (3.15)$$

The filter coefficients are generated using NASA's FDAS program. Substituting $z=e^{j\omega T}$ into (3.10) gives the amplitude response for the LPF shown in Fig. 3.7.

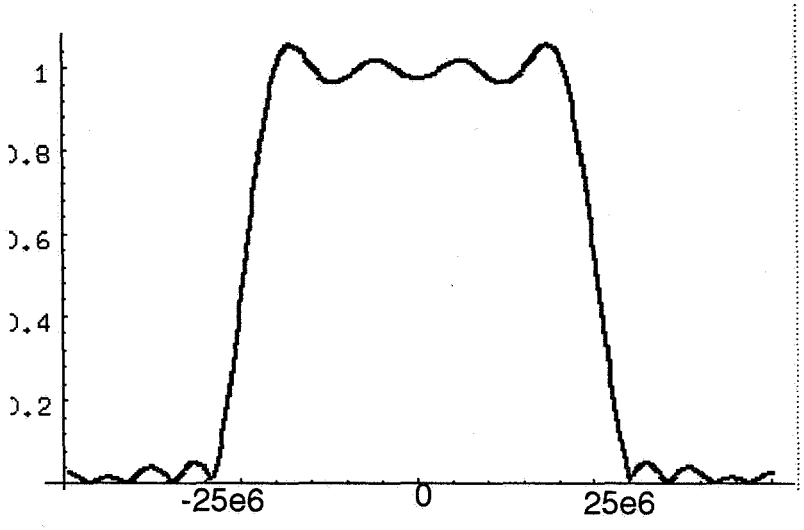


Fig. 3.7 Amplitude Response of the LPF

The filter structure that will implement equation (3.14) is shown in Fig. 3.8.

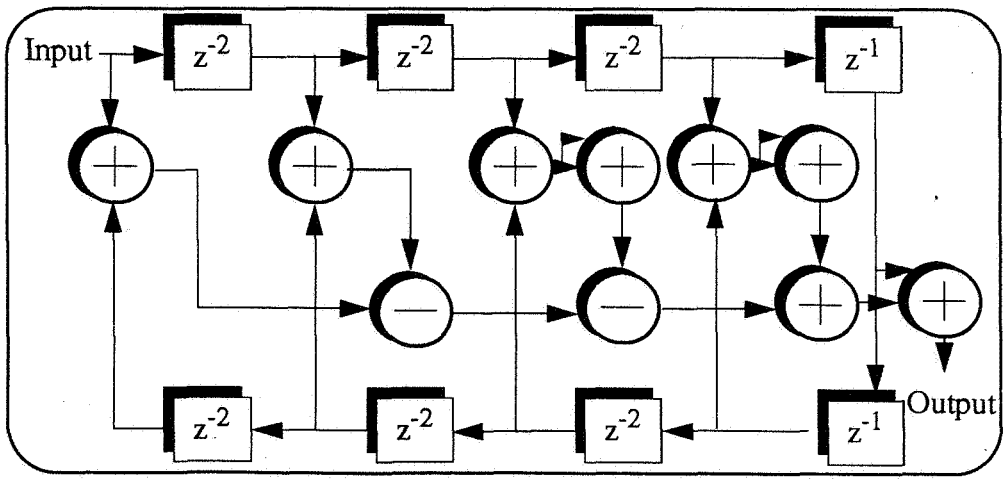


Fig. 3.8 LPF Architecture

This filter is designed with powers of two coefficients [7,8], so that there is no need

for area consuming multipliers. The multiplication can simply be done by shifts and add operations.

3.1.4 Integrate and Dump Unit

The integrate and dump unit (IDU) is the final stage of the correlation receiver. It's purpose is to integrate the input signal over one symbol period and make a decision as to what bit is sent. Since the rate of sampling is 4 samples per symbol, the IDU will accumulate four samples which is the same as integrating over a symbol period. This is demonstrated in the following equation:

$$g_{n+3} = \frac{1}{4}(f_{n+3} + f_{n+2} + f_{n+1} + f_n) \tag{3.16}$$

where g_n is the output of the IDU and f_n is the input samples to the accumulator. If the output of the IDU is positive, then the bit that is sent is assumed to be a logic 1 bit. If the output of the IDU is negative, then the bit that is sent is assumed to be a logic o bit.

To show that the IDU is a lowpass filter, the z-transform is performed on (3.16). Assuming that all initial conditions are zero, the z-transform of (3.16) is:

$$z^3 G(z) = \frac{1}{4}(z^3 F(z) + z^2 F(z) + zF(z) + F(z)) \tag{3.17}$$

The transfer function s

$$\frac{G(z)}{F(z)} = H(z) = \frac{z^3 + z^2 + z + 1}{4z^3} \tag{3.18}$$

If $z=e^{j\omega T}$ is substituted into (3.18), where T is the sampling period, the filter's amplitude response is found and is shown in Fig. 3.9.

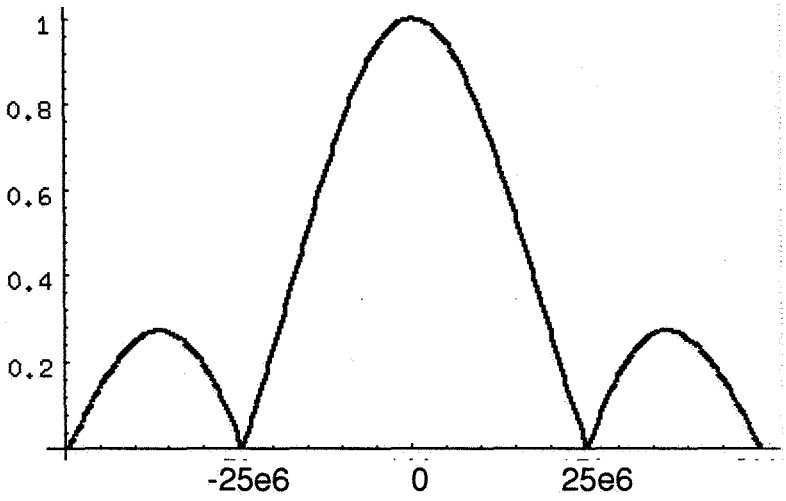


Fig. 3.9 Amplitude Response of the IDU

The IDU is shown in Fig. 3.10. The first latch is triggered by the 100MHz chip clock and the second latch is triggered by the symbol clock when it is time to make the decision s to what bit has been transmitted.

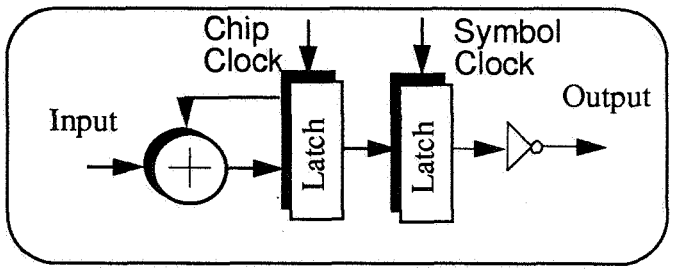


Fig. 3.10 IDU Representation

3.2 Phase Recovery Unit

The phase difference between bursts is proportional to the fame time:

$$\Delta\phi_{Frame} = 2\pi\Delta f T_{Frame} = (2\pi\Delta f T_{Frame})\frac{T_{Frame}}{T_{sym}} \qquad (3.19)$$

where Δ_{Frame} is the phase difference between bursts, Δf is the frequency uncertainty, T_{Frame} is the frame period and T_{sym} is the symbol period. If stable frequency sources are used then the Δf will be small making $\Delta f\, T_{sym} << 1$. But the frame time can be very long compared to the symbol time making $T_{Frame} / T_{sym} >> 1$, making the phase difference between bursts significant. Sine Δf is a uniform random variable, it makes the initial phase offset a uniform random variable in the range from $[-\pi, \pi]$ [9]. Therefore, the initial phase of the carrier must be estimated.

A phase recovery unit (PRU) for burst application using M-ary PSK has been proposed by Viterbi and Viterbi [9]. It is characterized by a fast acquisition time and is easily implemented with digital hardware. The PRU estimates the phase at the midpoint of an estimation interval which is composed of 2N+1 symbols, where N is the number of symbols before and after the symbol whose phase is being estimated. Initially the PRU takes 2N+1 symbols before it estimates the phase. After the first estimation interval, it needs only N more symbols to estimate the next phase because it re-uses the previous N symbols used in the previous estimation interval. This overlapping of the estimation intervals makes all estimates unbiased except for the first and the last estimate.

An M-ary PSK wave modulated with a burst technique is represented as:

$$s_i(t) = \sqrt{\frac{2E}{T_{sym}}} \cos\left(2\pi f_c t + \frac{2\pi i}{M} + \theta_{in}\right) \qquad (3.20)$$

where E is the symbol energy, T_{sym} is the symbol period, M is the number of unique symbols, i=0,1,..,M-1, f_c is the carrier frequency and θ_{in} is the initial phase of the carrier. First the carrier is input to a correlation receiver shown in Fig. 3.11.

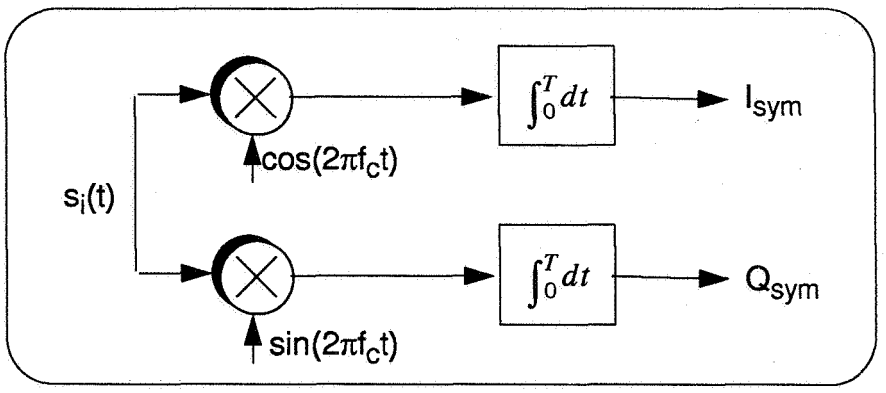


Fig. 3.11 Front End of the PRU

The inphase channel symbols will be obtained by multiplying the QPSK carrier by $\cos(2\pi f_c t)$ and integrating over one symbol duration:

$$I_{sym} = \int_0^T \cos\left(2\pi f_c t + \frac{2\pi i}{M} + \theta_{in}\right) \cos\left(2\pi f_c t\right) dt$$

$$= \int_0^T \cos\left(\frac{2\pi i}{M} + \theta_{in}\right) dt + \int_0^T \cos\left(2\pi 2 f_c t + \frac{2\pi i}{M} + \theta_{in}\right) dt$$

$$= \cos\left(\frac{2\pi i}{M} + \theta_{in}\right) \tag{3.21}$$

Similarly, the quadrature channel symbols will be obtained by:

$$Q_{sym} = \int_0^T \cos\left(2\pi f_c t + \frac{2\pi i}{M} + \theta_{in}\right) \sin\left(2\pi f_c t\right) dt$$

$$= \sin\left(\frac{2\pi i}{M} + \theta_{in}\right) \tag{3.22}$$

The next section of the PRU extracts the initial phase offset from the carrier. It views the two channels outputs in complex notation and does a rectangular to polar transformation on the quantity:

$I_{sym} + jQ_{sym} \Rightarrow \rho\, e^{j\theta}$

where

$$\rho = \sqrt{I^2_{sym} + Q^2_{sym}} \tag{3.23}$$

and

$$\theta = \text{atan}\left(\frac{Q_{sym}}{I_{sym}}\right) \tag{3.24}$$

The Viterbi algorithm requires that a non-linear transformation, T[ ], be performed on the magnitude ρ. According to Andrew and Audrey Viterbi, the non-linear transform should be:

$T[\rho] = \rho^k$

where k = 0,2,...M and even. For k=0, the performance of the PRU is best at E_b/N_o > 6dB. For k = 2, the performance is best at E_b/N_o < 0dB. Since k = 0 shows the best performance over a wider range of E_b/N_o, k is chosen to be 0 for this design. The phase θ must also be multiplied by M. The reason for this is demonstrated in the following equation:

$$e^{j\left(\frac{2\pi i}{M} + \theta_{in}\right)M} = e^{j2\pi i} e^{jM\theta_{in}} = e^{jM\theta_{in}} \tag{3.25}$$

This gets rid of all symbol phase information and leaves only the initial phase offset information. The magnitude and phase are then transformed back into rectangular coordinates:

$$\rho^0 e^{jM\theta_{in}} = \cos(M\theta_{in}) + j\sin(M\theta_{in}) \tag{3.26}$$

This complex notation is then split back into two signals:

$I_{snew} = \cos(M\theta_{in})$ and

$Q_{snew} = \sin(M\theta_{in})$

where I_{snew} and Q_{snew} are the new symbols in the inphase and quadrature channels respectively after the transformations. These symbols are then fed into an averager where they will be averaged over the estimation interval of 2N+1 symbols:

$$I_{av} = \frac{1}{2N+1}\sum_{i=0}^{2N} I_{snewi} \tag{3.27}$$

$$Q_{av} = \frac{1}{2N+1}\sum_{i=0}^{2N} Q_{snewi} \tag{3.28}$$

At this point the initial phase can be estimated. Since the phase is multiplied by M, the phase estimate will have to be divided by M:

$$\theta_{est} = \frac{1}{M}\operatorname{atan}\left(\frac{Q_{av}}{I_{av}}\right) = \frac{1}{M}\operatorname{atan}\left(\frac{\sin(M\theta_{in})}{\cos(M\theta_{in})}\right) = \frac{1}{M}M\theta_{in} = \theta_{in} \tag{3.29}$$

Assuming that I_{snew} and Q_{snew} remain approximately the same throughout the estimation interval. Since the atan() function tun result from [-π,π], the 1/M atan() will return a result from [-π/M,π/M]. This M-fold phase ambiguity needs to be taken care of since the incoming signal can be received with an initial phase in the range of [-π,π]. A unique word detector is used to resolve this ambiguity and will be described n a later section.

For this demodulator, M=4 since there are 4 unique symbols in both the QPSK/SQPSK modulation schemes. The number of symbols before and after the symbol whose phase is to be estimated is N=8. This corresponds with the number of symbols in [9] and provides adequate results. A brief summary of the algorithm is given below:

1.) Sum four consecutive samples (one symbol) in each of the channels to get I_S and Q_S.

2.) Do a rectangular to polar transformation to obtain the form: $I_S + jQ_S \Rightarrow re^f$

3.) Multiply the phase by four and do a nonlinear transformation T[] on the magnitude: $T[\rho] = \rho^0 = 1$, and $\phi' = 4\phi$

4.) Do a rectangular to polar transformation to get the new I and Q channel data.

$$\rho^0 e^{4\phi} \Rightarrow I_{snew} + jQ_{snew}$$

5.) Average I_{snew} and Q_{snew} samples over 17 symbols to get I_{av} and Q_{av}.

6.) The phase estimate will be:

$$\phi_e = \frac{1}{4}\text{atan}\left(\frac{Q_{av}}{I_{av}}\right)$$

Steps 2-4 can be implemented by storing the output values into a ROM and using it as a look-up table. Step 6 can also be implemented with a separate ROM. A block diagram of the PRU is shown in Fig. 3.12.

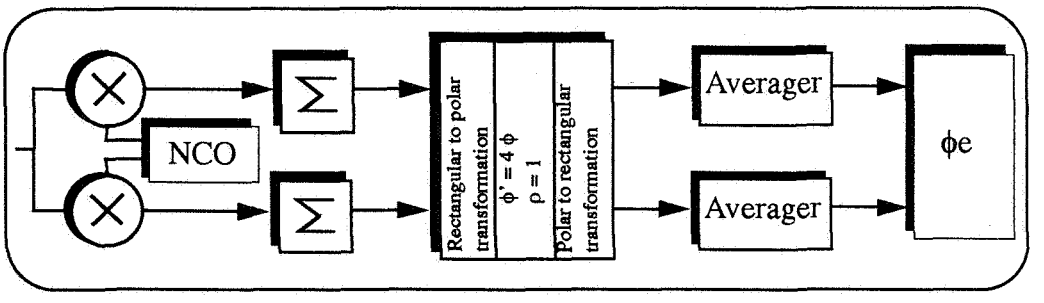


Fig. 3.12 PRU Block Diagram

3.3 Timing Recovery Unit

The timing recovery unit (TRU) [2] is needed to keep the receiver and transmitter synchronized in time. The symbol clock recovery takes place after the initial phase has been estimated. The clock is extracted by processing demodulated baseband waveforms. This is why the lowpass filters are needed in the correlation receivers. The TRU will use the TR bits in the preamble in order to initially lock on the timing. the number of TR symbols chosen for this design is 20

symbols of alternating logic bits. This TRU will reset the symbol clock after it has seen 3 consecutive alternating symbols.

The TRU uses the sign bits from the demodulated baseband signal to check for zero crossings such as those shown in Fig. 3.13.a.

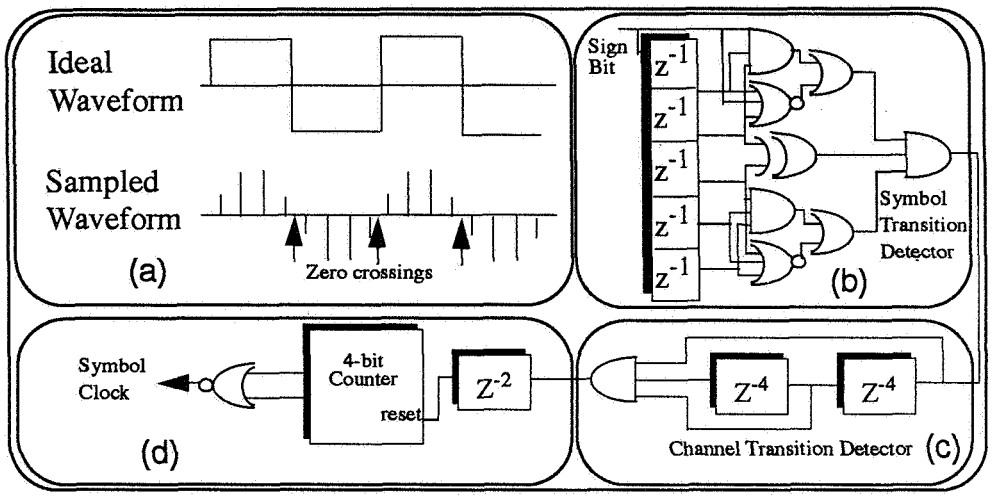


Fig. 3.13 Timing Recovery Unit

The number of sign bits, m, that can be used in the detection of the zero crossings are $2 \le m \le 2R$, where R is the number of samples per symbol that represent the signal. A larger value of m will result in a more accurate estimation of the zero crossing.

The first stage of the TRU, shown in Fig. 3.13.b. searches for zero crossings in the demodulated baseband signal. The sign bits of the demodulated signal are shifted into the shift registers whose outputs are used as inputs to the symbol transition detector(STD). The STD will output a transition signal when the first m/2 shift register bits are logic 1 and the last m/2 shift register bits are all logic 0 or when the first m/2 shift register bits are logic 0 and the last m/2 shift register bits are all logic 1 bits. This is demonstrated by the equation:

$$t_{n+m-1} = \prod_{i=0}^{\frac{m}{2}-1} s_{n+m-1-i} \prod_{i=0}^{\frac{m}{2}-1} \overline{s}_{n+i} + \prod_{i=0}^{\frac{m}{2}-1} \overline{s}_{n+m-1-i} \prod_{i=0}^{\frac{m}{2}-1} s_{n+i} \qquad (3.30)$$

where t_n is the zero transition sequence, s_n is the sign bits that are shifted through the shift registers, m is the number of sign bits used to detect the transition and n=0,1,... is the sample time. For this design, the number of sign bits used to determine the zero crossing is m=6. Equation 3.30 gives:

$$t_{n+5} = s_{n+5}s_{n+4}s_{n+3}\bar{s}_{n+2}\bar{s}_{n+1}\bar{s}_n + \bar{s}_{n+5}\bar{s}_{n+4}\bar{s}_{n+3}s_{n+2}s_{n+1}s_n \quad (3.31)$$

Equation 3.31 states that the transition will be a logic 1 when the first three sign bits are a logic 1nd the next three sign bits are a logic 0, or the first three sign bits are a logic 0 and the last three sign bits are a logic 1.

The channel transition detector (CTD), shown in Fig. 3.13.c, searches for a number of consecutive transitions before the symbol clock. This is to safeguard against any false transitions cased b noise. This operation is shown by:

$$c_{n+(r-1)R} = t_n\, t_{n+R}\, t_{n+2R} \cdots t_{n+(r-1)R} \quad (3.32)$$

where c_n is the channel transition sequence, tn is the zero crossing sequence from the STD, R is the number of samples per symbol and r is the umber of consecutive transitions that s required to reset the symbol clock where $r \geq 1$. The CTD for this design implements (3.32) with r=3 and R=4 which means that the CTD will wait for 3 consecutive zero crossings before it resets the symbol clock. This leads to (3.33).

$$c_{n+8} = tn\, t_{n+4}\, t_{n+8} \quad (3.33)$$

The symbol clock unit is shown in Fig 3.13d. The symbol clock will point to the first sample of the symbol as the counter counts 0.

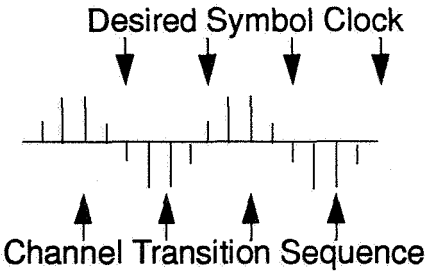


Fig. 3.14 Delaying of the Channel Transitions

The CTD will output a zero detection on the third sample of the symbol. This is illustrated in Fig. 3.14. For this reason, the CTD's output needs to be delayed by 2 sample time units so that it will rest the clock to 0 at the time of the first sample of the symbol.

3.4 Unique Word Detection

A unique word detector (UWD) [10] utilizes the UW bits in the preamble to tell when the data starts. The bits at the output of the inphase and quadrature correlation receivers are fed into the I channel UWD and Q channel UWD respectively. Since the QPSK signal can be received with a phase offset anywhere in the range of $[-\pi,\pi]$, the I channel bits or the Q channel bits or both may be inverted. For this reason it is desired to use a matched receiver that will have a minimum output when the bits in the shift registers matched the ones stored in the UW memory and a maximum when the bits in the shift registers re the inverse of the ones stored in the UW memory. The data starts when the UWD reaches either a maximum or a minimum. Fig. 3.15 shows both I channel and Q channel UWDs.

The Unique word (UW) was chosen as a random sequence of 15 bits and determined to have a false alarm probability of 10^{-8} which is within the target BER of the satellite. More discussion of the UW is in section 5.6.

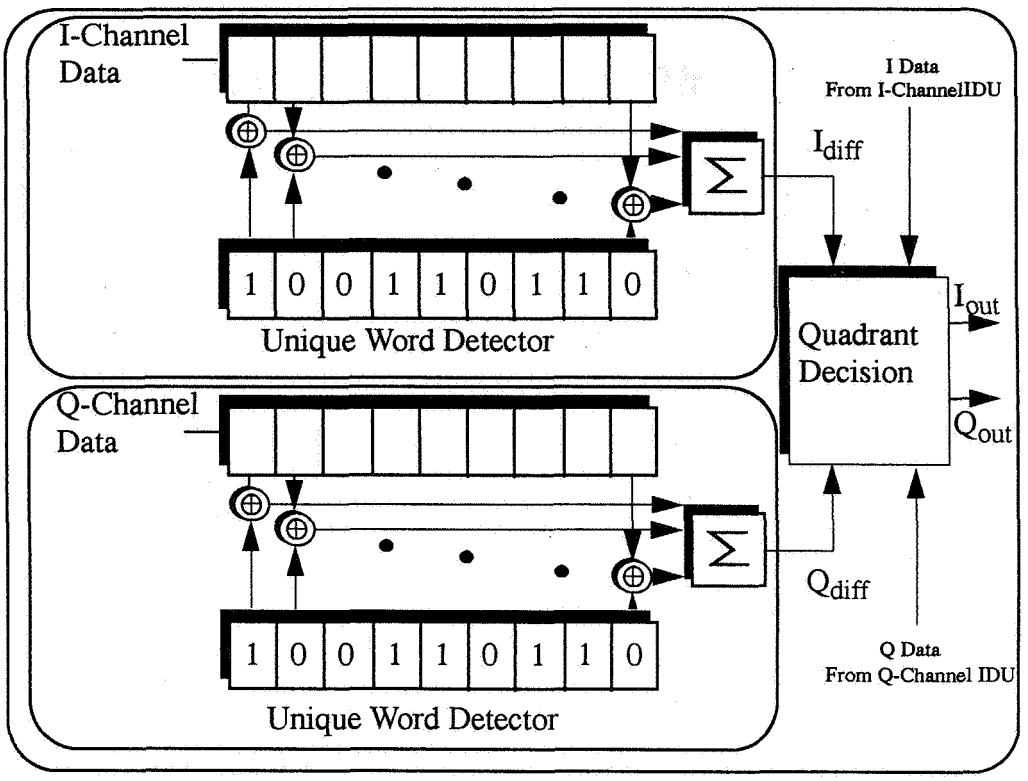


Fig. 3.15 Unique Word Detectors

The characteristics of the UWD lead to an easy way to resolve the phase ambiguity of the PRU. The outputs I_{diff} and Q_{diff} are the differences between the shift register bits and the unique word bits in the I and Q channel respectively. These values are obtained by XORing the shift register bits with the unique word bits and adding all of the XOR outputs. The quadrant decision unit (QDU) compares I_{diff} and Q_{diff} to a threshold value and maps the signal into the region between $[-\pi/4,\pi/4]$ as shown in Table 2.

Table 2: Quadrant Decision.

Condition	$Q_{diff} <= 5$	$Q_{diff} > 5$
$I_{diff} <= 5$	$I_{out} = I_{in}$ $Q_{out} = Q_{in}$	$I_{out} = \overline{Q}_{in}$ $Q_{out} = I_{in}$
$I_{diff} > 5$	$I_{out} = Q_{in}$ $Q_{out} = \overline{I}_{in}$	$I_{out} = \overline{I}_{in}$ $Q_{out} = Q_{in}$

As an example, let I_{diff} =0 and Q_{diff} = 9, which means that the inphase bits re received correctly and the quadrature bits are the inverted version of the correct bits. If the threshold value is 5, the QDU maps the inverted data from the Q channel IDU into Iout and the data from the I channel IDU into Qout. This in effect maps the signals from the quadrant that the signals are received in to the quadrant between $[-\pi/4,\pi/4]$, which is now in the range of the PRU.

Chapter 4 Generators

4.0 Design Considerations

From each of the architectures in the preceding chapter, a group of commonly used components can be extracted. These components are the binary adder, multiplier, read-only memory and serial access memory. Since many different sizes of these components are used, component generators are developed. The following sections illustrate some of the design considerations used when creating the generators and some background information on the generators developed for this project. For further information, see [11].

4.1 Area and Performance

Area is a very important design consideration when creating a VLSI system. The cost of the system is proportional to the area of the system, so it is advantageous to make the designs as small as possible. It is best to design a component with the dimensions of a square. Any other shape will result in costly space between the components as shown in Fig. 4.0.

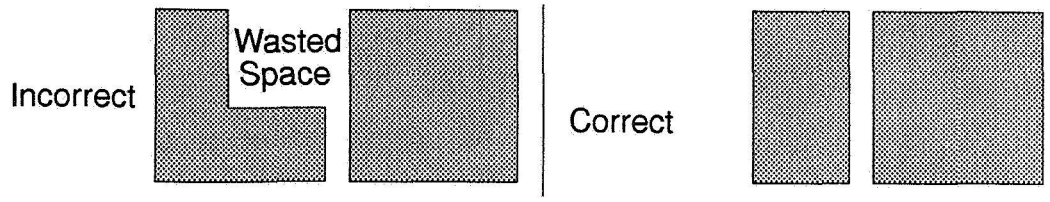


Fig. 4.0 Area Considerations

Minimization of the number of I/O pin connections to a component is a necessity. Smaller components should only have one power, ground and clock connection. This will minimize the amount of wiring that is needed to connect the component to the system and therefore, the area of the channels will be smaller. A template that should be used for all components is shown in Fig. 4.1.a. This will result in an efficient layout as shown in Fig. 4.1.b.

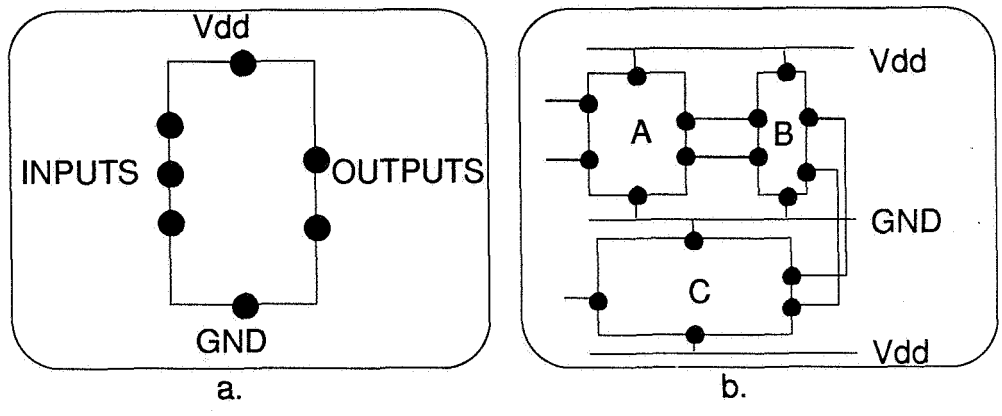


Fig. 4.1 Component Template

Performance is another major consideration when designing a VLSI system. The performance of the system heavily depends on the area of the system. Small area leads to smaller capacitance and faster, higher performing circuits. Performance is also based on how much power the system dissipates. Proper sizing of the transistors can alleviate the static power dissipation and minimize the dynamic power dissipation in the CMOS technology.

4.2 Hierarchy

Hierarchy is the process of breaking down a system into smaller components in order to create a less complex design. Instead of trying to design a large system all at once, it is easier to design the smaller sub-components first

and then put these components together to create the larger ones. The demodulator is divided into seven main components. The larger of these components are divided even further.

4.3 Regularity

Regularity decreases the complexity of a system by dividing the hierarchy into a set of similar building blocks. In other words, it uses specific designs in a number of places, which is where the generators play their important part. Using the same component more than once allows for quick designs as well as less masks to be produced, which reduces the cost of the system.

4.4 Modularity

The hierarchical designs should interface with one another cleanly and precisely which is why modularity is of great importance to a system. The interconnects between two components should be at the same coordinates and on the same level of metal in order to avoid the addition of costly contacts and space. When all of the components are connected together, the dimensions should be that of a square in order to obtain a higher level of integration. Modularity can also be expressed in the timing among the components. It is necessary that the operations between two connected components be precisely timed or this will decrease the performance of the system.

4.5 Generators

The layout of regular structures such as adders, ROMs, and multipliers may be synthesized by software generators. These programs take a

number of parameters as input and automatically create a custom physical layout. Efficient algorithms are used to translate the input parameters into the output descriptions. The key reasons for the development of generators is that they assure a shorter design cycle and allow for more exploration of different design styles since they can be generated quickly and easily.

The fixed cell approach has been used in the development of the generators for this research. This means that all of the cells are of fixed size and can not be changed. A cell library can be developed which contains many different sizes of these cells. The generator can be designed to call up different cells for different input parameters. All cells are made arrayable, which means that when the cells are appended together, the proper terminals will be connected.

Mentor Graphics Led is the main building block used in the development of the generators. All cells in the cell library are created using Led. A programming language called Lx is a procedural interface to the L database and Led graphics editor. Lx provides access to the information within the L database and provides interaction with the Led graphics editor. This allows Lx to accept an input from the Led command prompt, call up the necessary cells from the cell library, and place the cells according to the algorithm. Since Lx can read from the database, it can be used to wire components, place contacts where needed and place top level connectors automatically.

4.6 Adder Generator

Addition is the fundamental operation in digital systems. It is used in adders, subtracters, multipliers and many other applications. Of the many different types of adder cells, the transmission gate adder is chosen for this research. This cell is only made up of 24 transistors which make the area and performance very

attractive. The transmission gate adder is shown in Fig. 4.2.

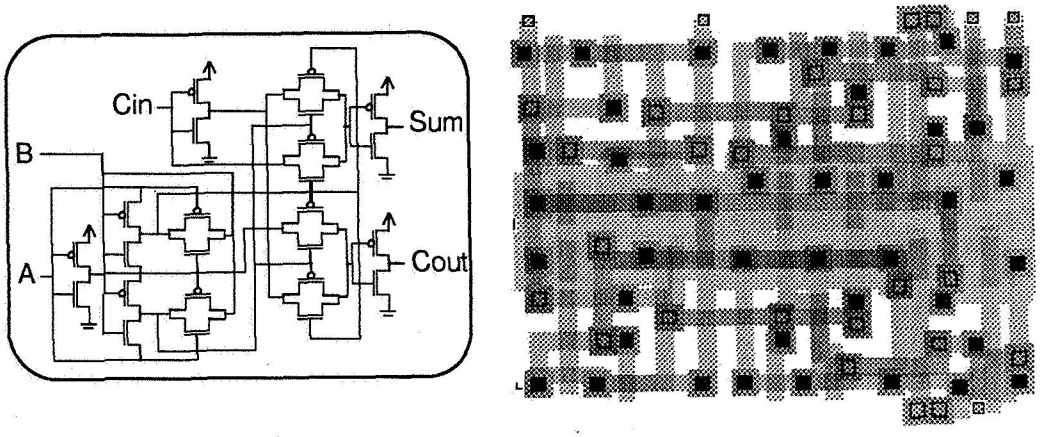


Fig. 4.2 Transmission Gate Adder Cell

The transmission gate adder cell can be used to add two 1-bit numbers. In order to create an n-bit adder, this cell can be cascaded n times to form a ripple carry adder shown in Fig. 4.3. Instead of appending the adder cells into one long adder, the generator folds the adder in half in order to conserve on area.

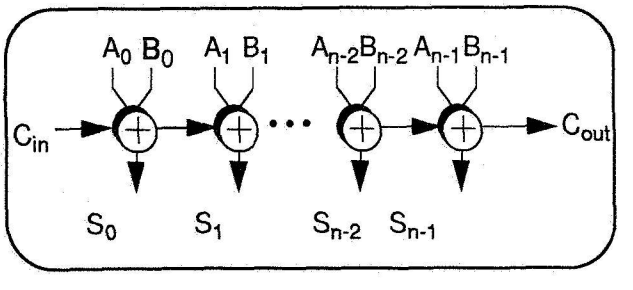


Fig. 4.3 N-bit Adder

The demodulator is composed of many different sizes of adders. An adder generator is developed in order to cut down on the design time of each of these

adders.

4.6.1 Basic Cells

There are four different cells that are used by the adder generator. All of these cells are similar to the one shown in Fig. 4.2. There are two main design concerns when creating this cell. First, the connectivity between the cells must be very clean and compact. The area is then minimized and the adder is as compact as possible. Second is to design the cell such that the SUM and CARRY signals propagate at the same time. The output of each stage depends on both the SUM and the CARRY of the preceding stage. The critical path is the CARRY signal, so it is important to minimize this delay such that it propagates at the same time as the SUM signal.

The simulation of the adder cell is done in the adept mode in order to generate the true characteristics of the adder. Fig. 4.4 shows that the SUM and the CARRY signals of the loaded adder cell propagate at the same time as anticipated.

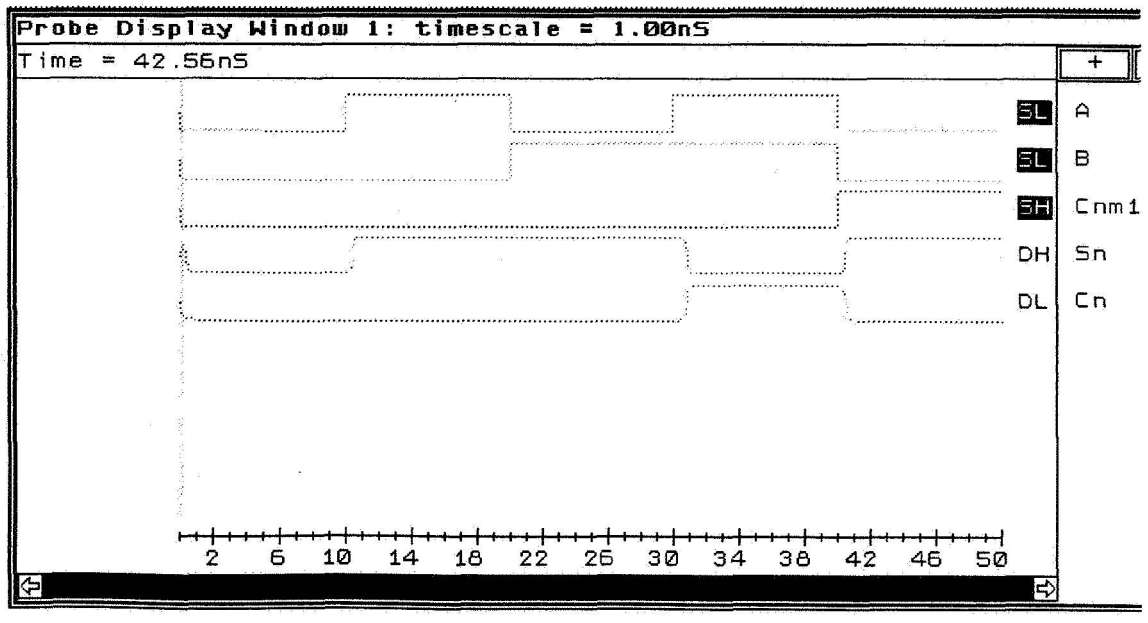


Fig. 4.4 Simulation of the Basic Cell

4.6.2 Algorithm

The adder generator creates the n-bit adder according to the flow chart in Fig. 4.5. The size and name of the adder are input parameters supplied by the user. The input parameters basically specify the organization of the adder. The algorithm places the top adder cells followed by the bottom adder cells and the layout is created. The automatic wiring and top level terminals are then placed. This generator places a constraint on the size of the adder. The user must specify only even sized adders such as an 8-bit or a 6-bit adder. This greatly reduces the complexity of the generator as well as decreases the wasted space caused by an odd number of adder cells.

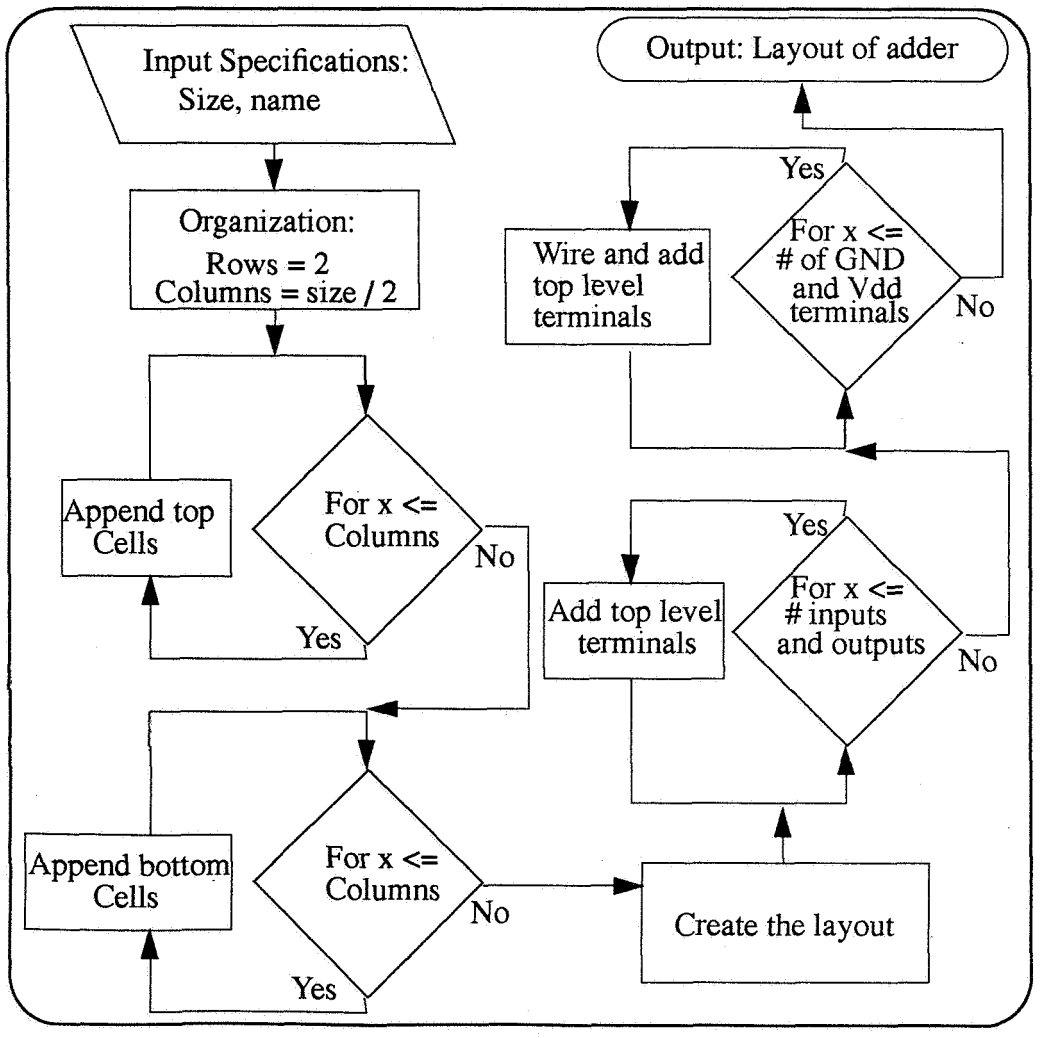


Fig. 4.5 Adder Generator Algorithm

4.6.3 Output

The generator output of an 8-bit adder is shown in Fig. 4.6. It is a folded architecture with inputs on both sides and outputs on one side.

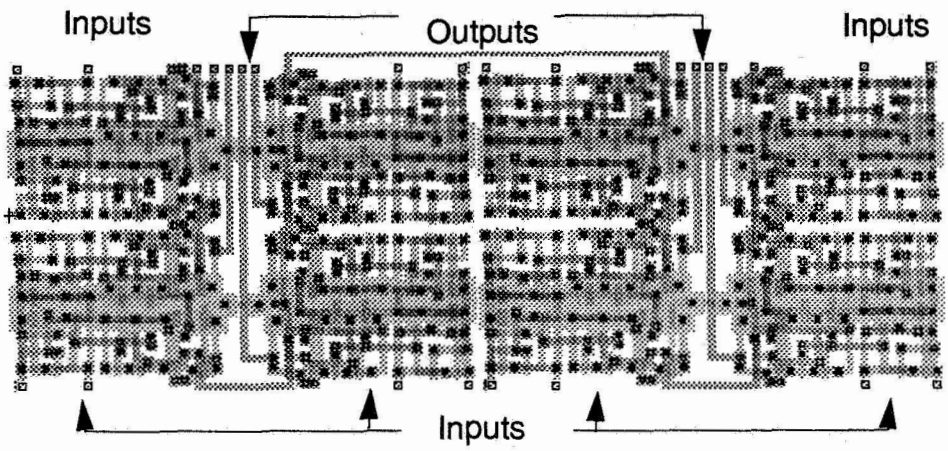


Fig. 4.6 8-bit Ripple Carry Adder

The area of this adder is 140 x 50 μm^2, and the worst case delay is approximately 6 ns. This is well in the range of the 10 ns clock period. Fig. 4.7 shows an adept simulation of the 8-bit adder where a_{in} and b_{in} are the inputs and sout is the output.

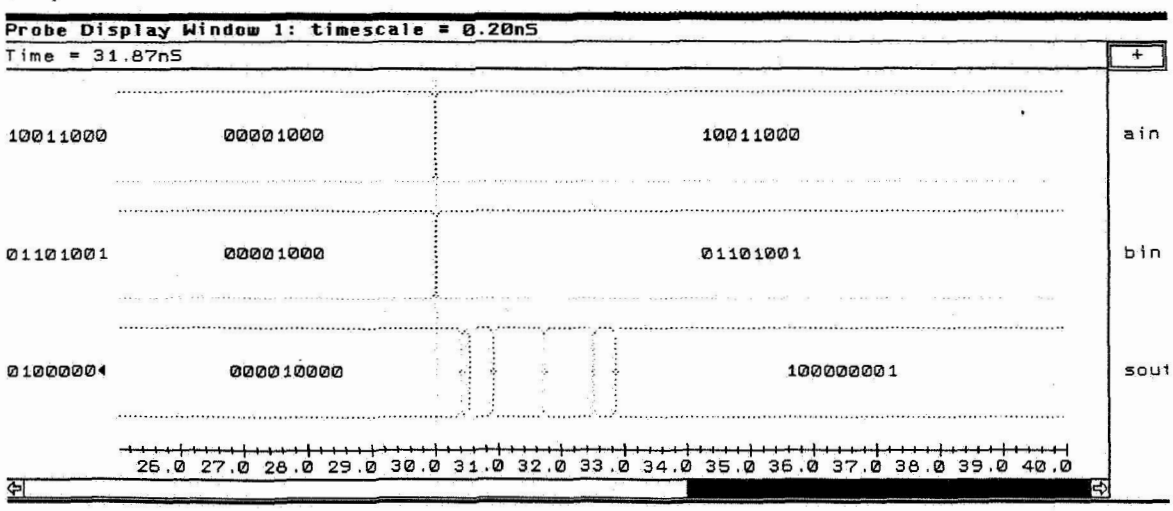


Fig. 4.7 Simulation of 8 Bit Adder

4.7 Multiplier Generator

Binary multiplication is based on a technique of successive additions and shifts in which each addition is conditional on one of the multiplier bits. This allows the multipliers to be made with adder cells. The adder cell used for the multiplier is the same as the adder cell used in the adder generator with a few modifications. The Baugh/Wooley multiplication algorithm is used[7]. This algorithm is designed for high speed, two's complement multiplication. The main advantage of this algorithm is that the signs of all the partial product bits are positive, allowing the product to be formed using array addition techniques. A modified version of the Baugh/Wooley algorithm is used which saves three adder cells over the original algorithm [8].

4.7.1 Basic Cells

The multiplier is designed with the same basic cell as the adder, except for one simple modification. There is an AND gate added to the layout of the adder cell. The multiplier cell block diagram is shown in Fig. 4.8.

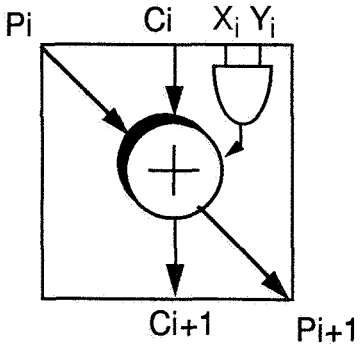


Fig. 4.8 Multiplier Cell Block Diagram

The four basic cells are shown in Fig. 4.9. The AND gate is used to do the first row

of multiplications. After this stage, the shift and add technique is carried out by the multiplier cells. The pipelining stage is composed of serial access memory cells. The final stage is made up of adder cells without any AND gates.

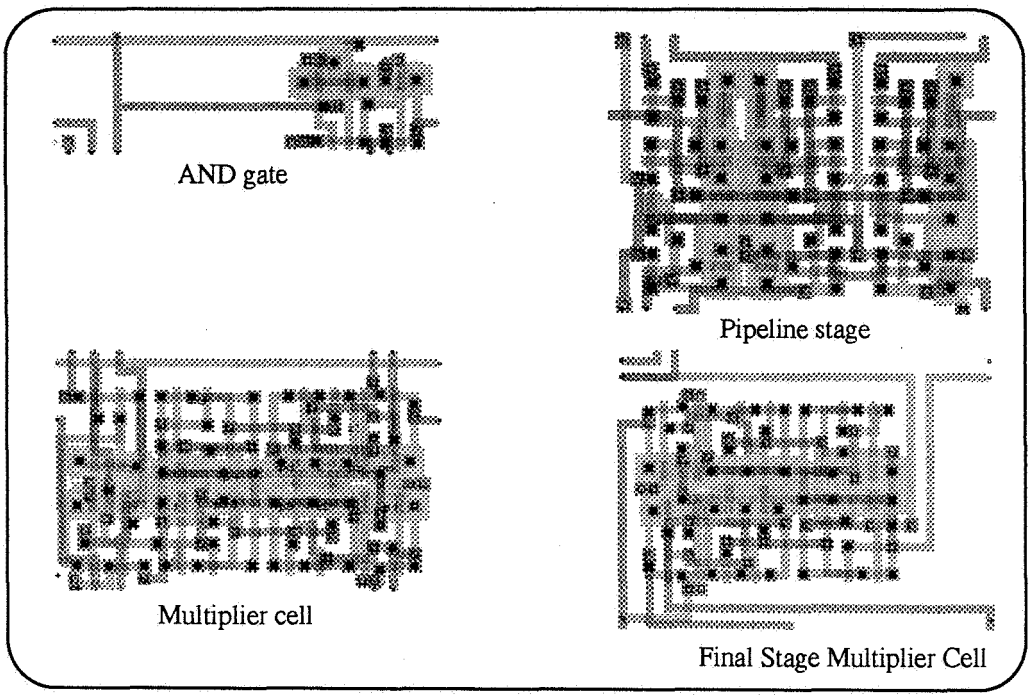


Fig. 4.9 Multiplier Basic Cells

4.7.2 Algorithms

The flow chart in Fig. 4.10 illustrates the algorithm of the multiplier generator. The inputs specify the organization of the cells. First the algorithm places the AND gates, then the multiplier cells. The multiplier is pipelined so the pipeline stage is placed between the multiplier cells. The final stage of adders is then placed and the multiplier is generated.

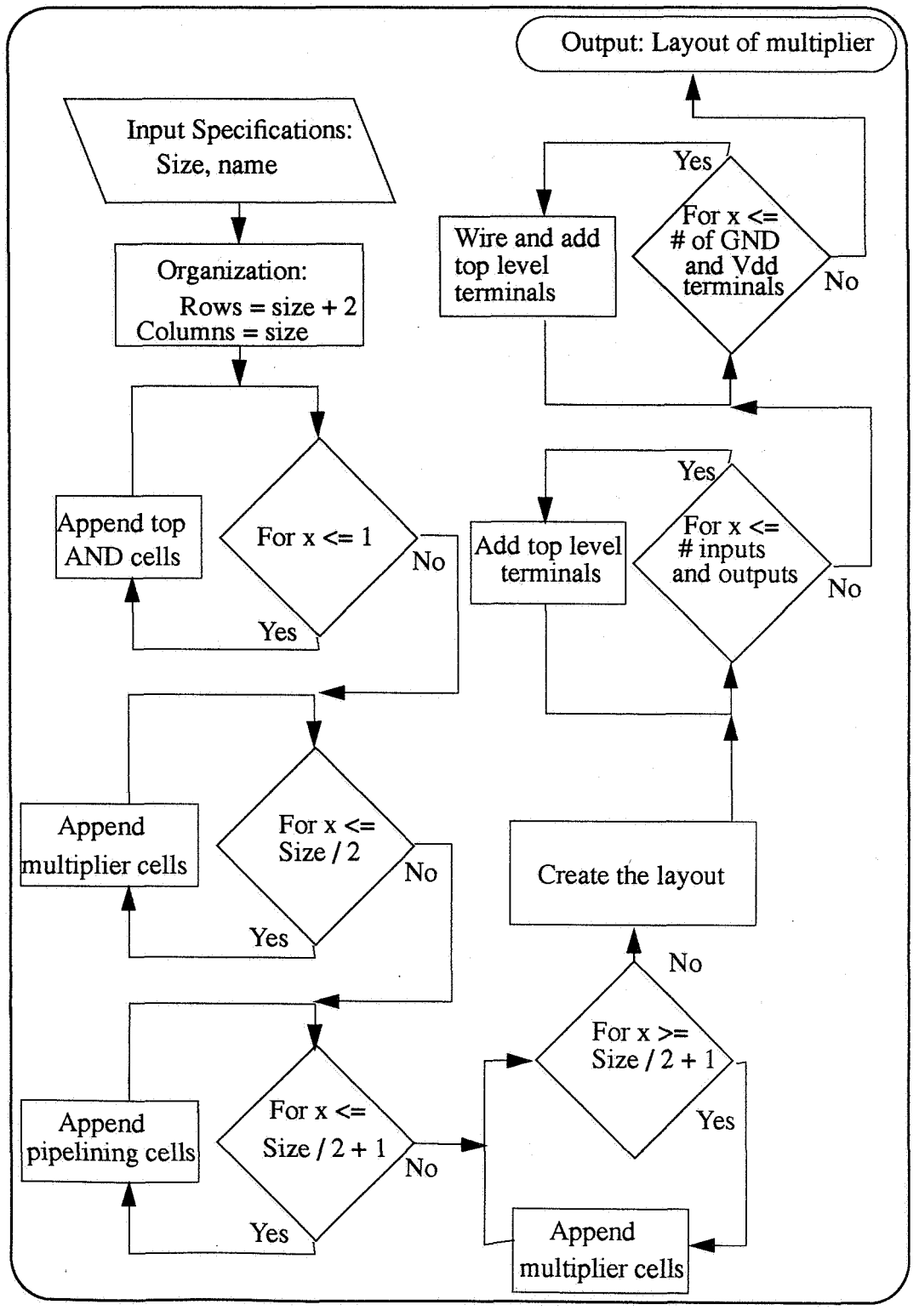


Fig. 4.10 Flow Chart for Multiplier

4.7.3 Output

The generator output of an 8-bit by 8-bit multiplier is shown in Fig. 4.11. There is only one VDD terminal and one GND terminal, on the MET3 level, needed to be connected to this component. All of the outputs come from one at the MET2 level. The inputs are on both sides at the MET1 level.

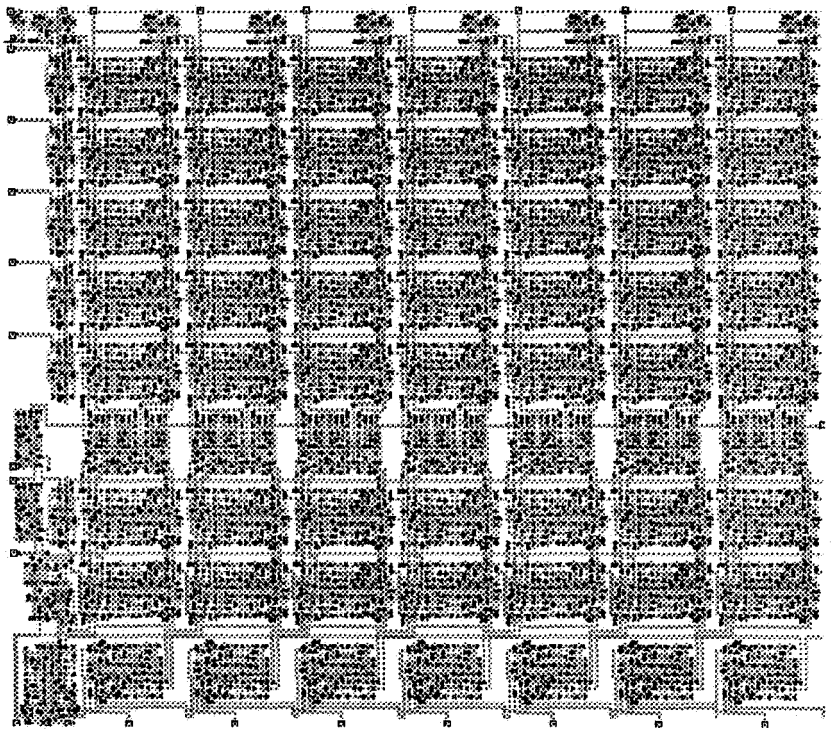


Fig. 4.11 Output of Multiplier Generator

The simulation in adept mode of all of these transistors is beyond the capabilities of the simulator Lsim. So another means is used to simulate the multiplier. The delay information from each of the layout cells is extracted and saved in a file. Each cell is very accurately represented in VHDL and the delay information is read from the file as a look up table. The VHDL cells are then placed together and simulated. The results showed that the worst case delay for an 8 x 8 bit multiplier is about 12 ns. The multiplier had to be pipelined because the clock period is only

10 ns.

4.8 Read Only Memory

The read-only memory (ROM) is a static memory structure. The memory is programmed in during fabrication and can never be changed afterwards. When algorithms require many mathematical computations, ROMs can be used as look-up tables in order to increase the computing capability of the system. The ROM will give a faster output and reduce the system complexity at the cost of more area used.

Since it would be impossible to design a large ROM by hand in an efficient amount of time, a ROM generator is developed. This generator is composed of three smaller generators: the row decoder generator, the ROM array generator, and the bus generator.

4.8.1 Organization and Operation

The organization of the ROM is shown in Fig. 4.12.

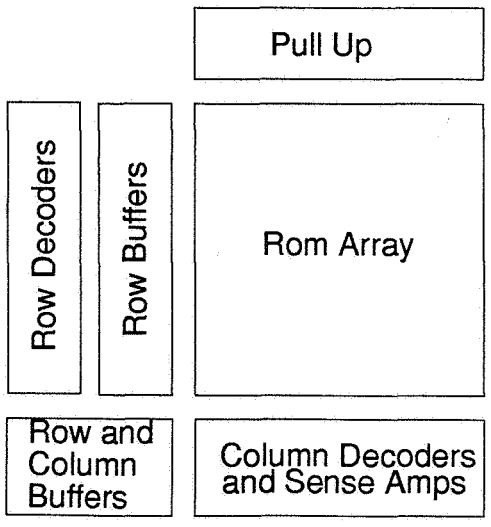


Fig. 4.12 Organization of the ROM

The inputs are fed through the row and column buffers and into the row and column decoders. Then the decoders choose which location of memory is to propagate to the output.

For proper operation of the ROMs, two clocks are needed. The operation of the ROM is as follows:

1.)The pull up circuits pre-charge the bit lines and the row decoders pre-discharge the word lines.

2.)Depending on the input, the row decoder will conditionally charge a word line and the column decoder will choose a column. The bit line will be discharged if a transistor has been selected, or it will remain charged.

3.)The sense amp is enabled.

4.)The data is latched.

The clocking scheme is illustrated in Fig. 4.13.

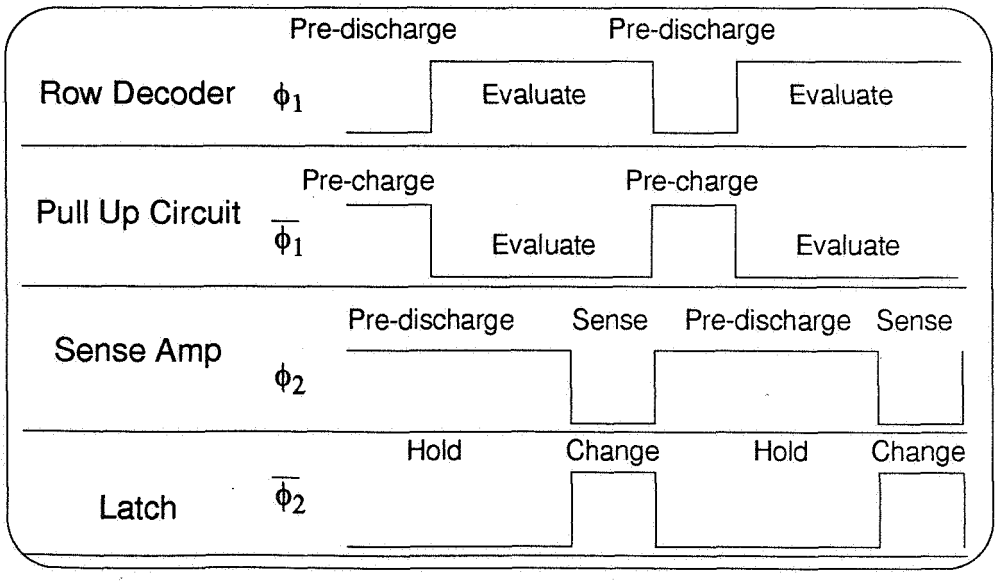


Fig. 4.13 Rom Clocking Scheme

4.9 Row Decoder Generator

A row decoder is used to select one word line out of 2^n word lines, where n is the number of inputs to the row decoder. The row decoder designed for this research is an AND based tree decoder. This type of decoder is very regular and very easily generated. Fig. 4.14 shows a diagram of the tree decoder.

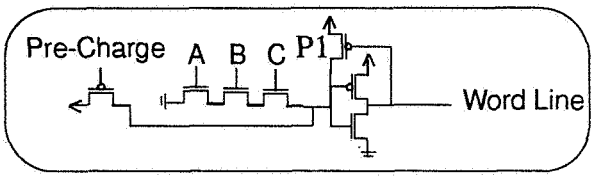


Fig. 4.14 AND based Tree Decoder

It is based on a dynamic tree decoder where the output is first pre-discharged and then conditionally charged. A modification has been made to the dynamic decoder in order to make it static. The p-transistor P1 has been introduced to the circuit for this purpose. When the word line is discharged, the p-transistor turns on and guarantees that the input to the inverter will remain high. This is very important when row decoding since any leakage current could cause an incorrect word line to go high. When the word line is selected, the output of the inverter will go high and the p-transistor will turn off. The row decoder pre-discharges the word line so that no word in the memory is selected. Since the word line is low, the p-transistor turns on and keeps the input of the inverter high. Inputs A, B and C conditionally turn the n-transistors on and discharge the input of the inverter. The word line goes high selecting a row of memory and turning off the p-transistor.

4.9.1 Row Decoder Basic Cells

The basic cells for the row decoder are shown in Fig. 4.15. The

transistors are placed such that each row has a different address. Therefore, there is a need for a cell with a transistor and a cell with no transistor. The pre-charge stage is the last cell appended to a row. Each row needs to be pre-charged which will pre-discharge the word lines.

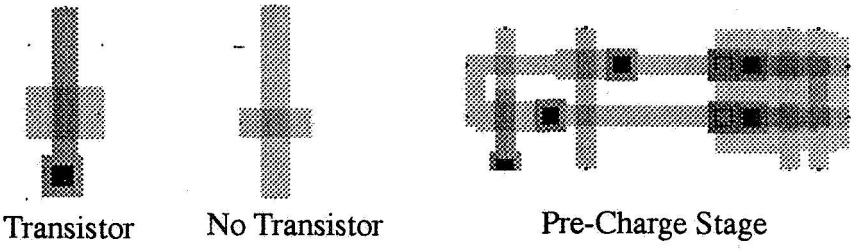


Fig. 4.15 Row Decoder Basic Cells

4.9.2 Algorithm for the Row Decoder

The algorithm for the row decoder is shown in Fig. 4.16. The address table is automatically generated at the beginning of the program. It provides address information for each of the rows and tells when to append a transistor cell or a cell with no transistor. The pre-charge circuit is appended at the end of each of the rows.

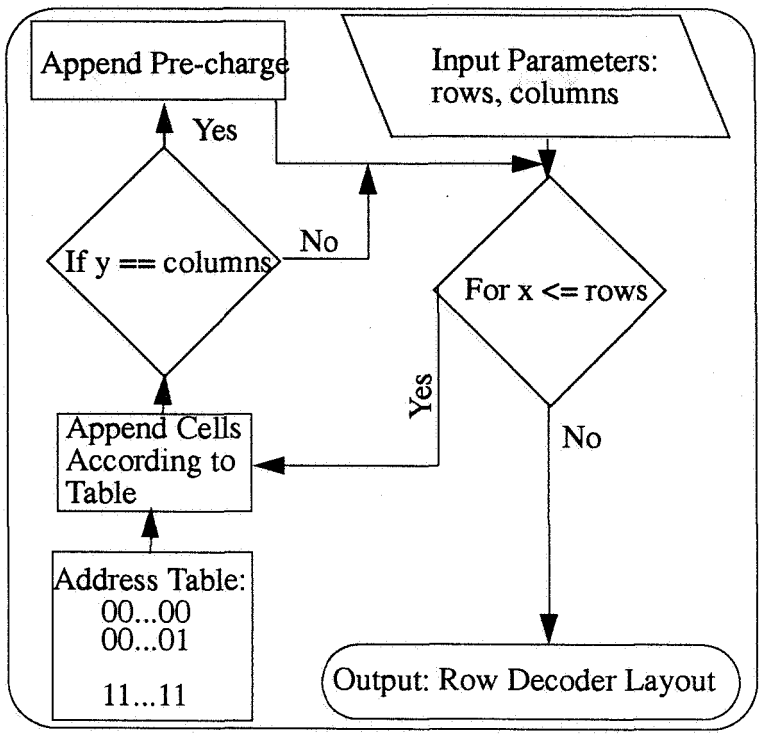


Fig. 4.16 Row Decoder Algorithm

4.9.3 Output

The output of the row decoder is shown in Fig. 4.17.

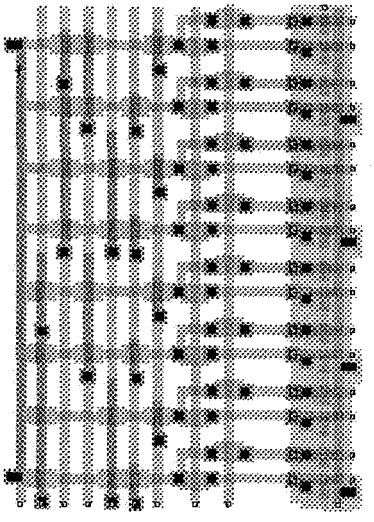


Fig. 4.17 Decoder Generator Output

The simulation of the row decoder with the output buffer stage is

shown in Fig. 4.18. This decoder has 16 rows to choose from. To show the functionality of the row decoder, each of the rows is selected one after the other.

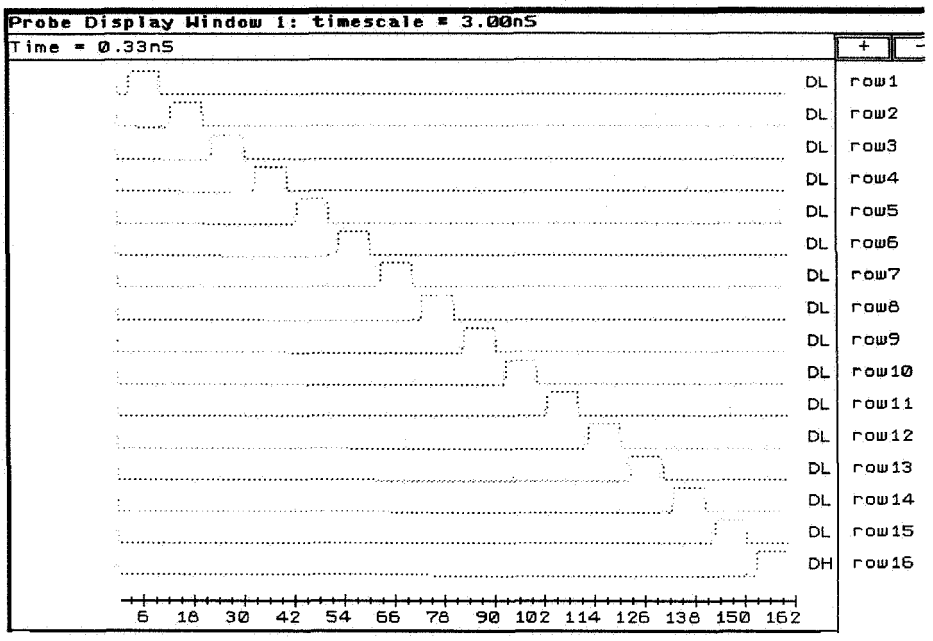


Fig. 4.18 Simulation of the Row Decoder With Buffered Output

4.10 ROM Memory Array

The ROM array is composed of pre-charge circuitry, memory cells and pass transistors. The pre-charge circuitry, shown in Fig. 4.19, is made up of load transistors and pulsing transistors. During pre-charge, the pulsing transistors and the load transistors charge up the bit lines. During the evaluation period, the pulsed transistors are turned off to save on power dissipation, but the load transistors are still on. The load transistors are very weak and allow only a small amount of current to flow. This is to defeat any leakage current that may be discharging the bit lines. The current through the load transistor is also small enough such that the ROM array transistors can easily pull down the bit line. The load transistors will not allow the ROM array transistors to pull the bit line all of the way down to 0V, which will speed up the pre-charge operation. The bit lines are only charged up to about 3.4V because the pull up transistors are all n-transistors.

An n-transistor pre-charge circuit saves area over a p-transistor pre-charge circuit and the pass transistors at the end of the ROM array only allow 3.4V through anyhow, so there is no degradation in performance.

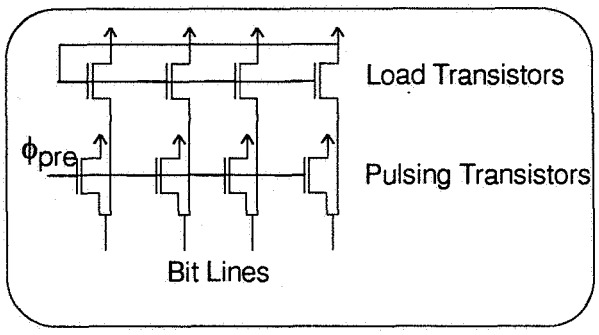


Fig. 4.19 Pre-Charge Circuit

The ROM array is composed of the memory transistors. Each bit of memory requires one n-transistor. If a logic 1 is needed to be programmed into the memory, no transistor is placed on the bit line. When the word line is selected, then there is no transistor to discharge the bit line, and it will remain logic 1. If a logic 0 is needed to be programmed into memory, then a transistor from the bit line to ground is placed in the array. Now when this transistor is selected by the word line, it will discharge the bit line to a logic 0. The array is designed to be as compact as possible. There are many design techniques used to increase the performance of the ROM array. Contacts are shared as much as possible to save on area and bit line capacitance. A technique called strapping is done every eight transistors on the polysilicon word line and on the n-diffusion ground line. Strapping is done by placing a low resistance metal line over the top of a high resistance line and connecting these two lines every eight to sixteen transistors. This places a large resistance in parallel with a low resistance and the equivalent resistance is smaller than the smallest resistance. The ROM array schematic is shown in Fig. 4.20.

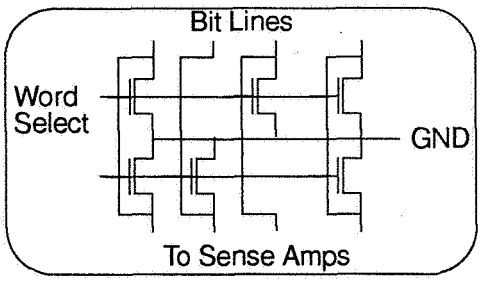


Fig. 4.20 ROM Array

4.10.1 Basic Cells

The basic Cells for the ROM memory array are shown in Fig. 4.21. The pre-charge circuitry is shown with the load and pulsing transistors.

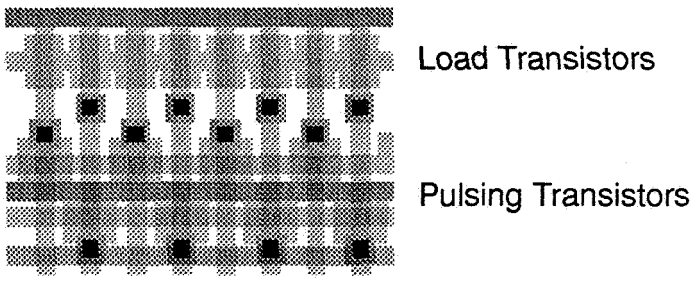


Fig. 4.21 Pre-Charge Cells

The cells in Fig. 4.22. are the memory cells. These are divided into two groups: top cells and bottom cells. The top cells are placed in the odd rows and the bottom cells are placed in the even rows. There are three different types of top cells. The cell with no transistor is used when a logic 1 is programmed into memory. There are two types of cells with no transistor. The one with a contact is placed when there is a transistor above it. These two transistors share the contact which saves on area. The one with no contact will be placed when there is no contact above it.

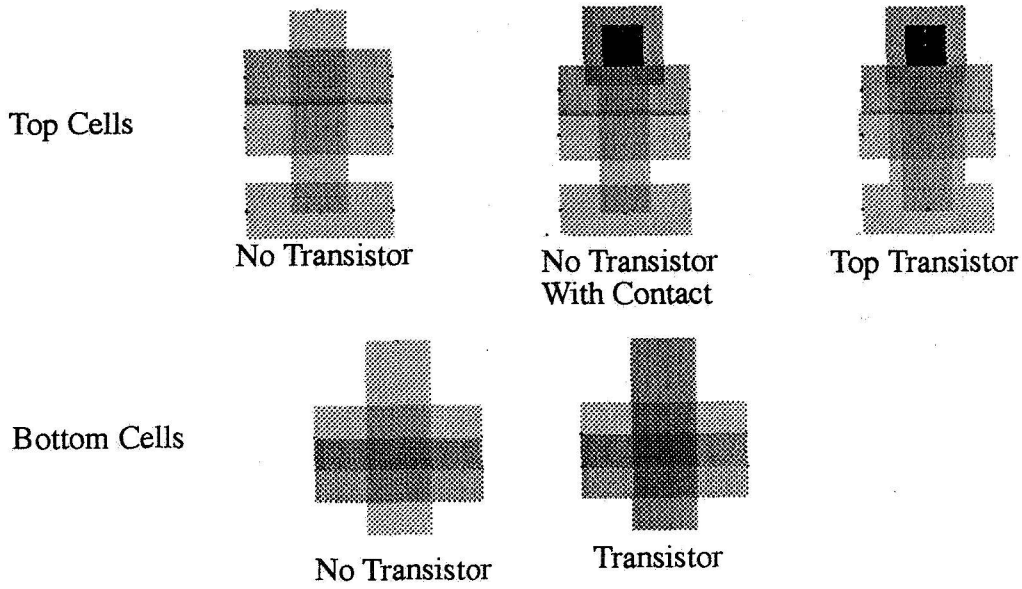


Fig. 4.22 ROM Array Basic Cells

The pass transistor, shown in Fig. 4.23, is used in the final stage of the memory array. These transistors are placed at the end of each of the bit lines to act as a switch. The column decoder selects a group of pass transistors, enabling the logic level on the bit lines to be sensed by the sense amps.

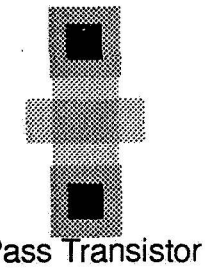


Fig. 4.23 Pass Transistor

4.10.2 Algorithm

The ROM array generator first appends the pre-charge transistors. Next it starts placing the memory transistors according to the data file. The data file does not have to be in any particular format. It can be contained in a text file in columns or in a single array of numbers. The input specifications will tell the ROM array generator how many bits to read from the file. After the memory transistors are all in place, the pass transistors are then appended. The layout is then created. The algorithm is illustrated in Fig. 4.24.

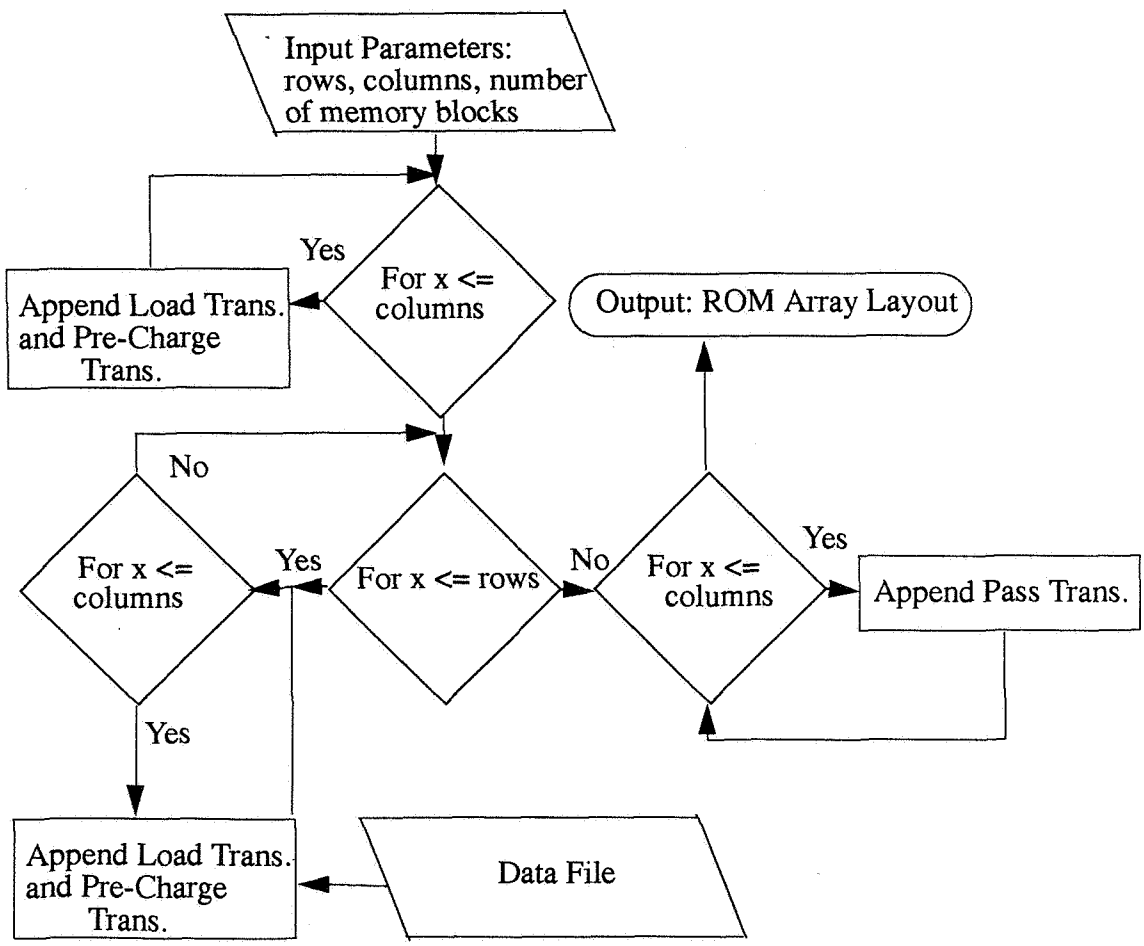


Fig. 4.24 ROM Array Algorithm

4.10.3 Output

An example ROM is shown in Fig. 4.25. The row decoder, row buffer, pull up circuit and ROM array are all generated and automatically placed together. The input buffers, bus, column decoder and sense amps are all placed by hand.

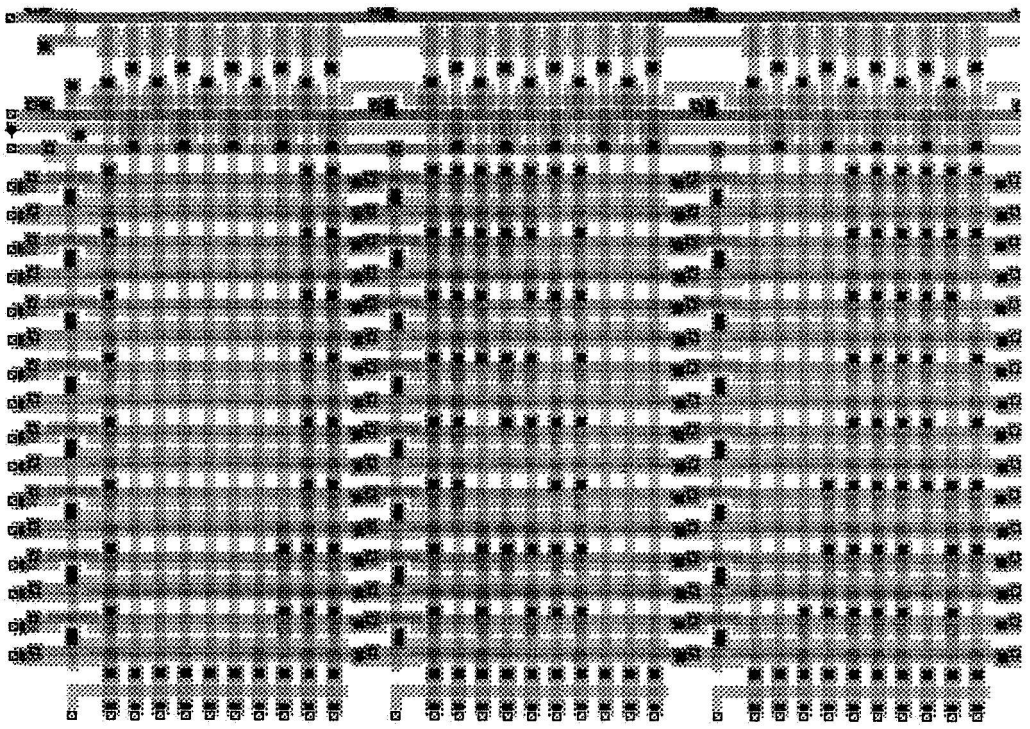


Fig. 4.25 ROM Array

4.11 ROM Generator

The generators that have been developed thus far produce pieces of a ROM. Another generator is needed to integrate these pieces into one unit. This is the ROM generator. The ROM generator developed for this research is only a partial ROM generator. It places the row decoder, the row buffers and the ROM array circuits and automatically connects these units together. The input

buffers and the bus are placed in the ROM cell by hand.

4.11.1 Output

Fig. 4.26 shows an example of a 16 x 32 bit ROM. This particular ROM is only for demonstration purposes and is not used in the research.

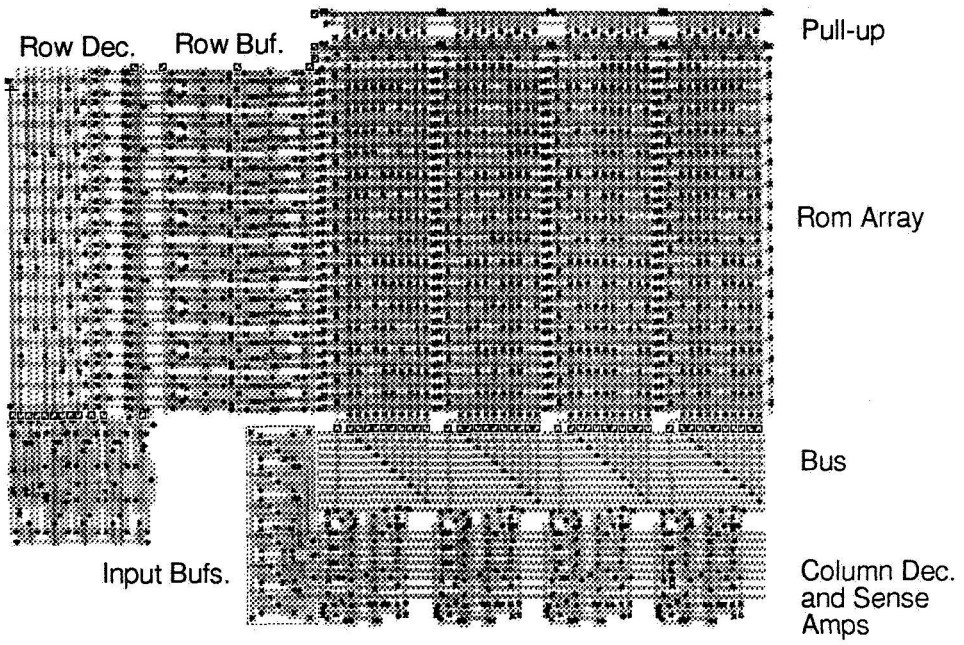


Fig. 4.26 Output of the ROM Array Generator

4.11.2 Bus Generator

In a memory system composed of n-bit words, it is advantageous to use only n sense amps. Therefore, when the memory is split up into multiple blocks, it is necessary to connect all of the first bits of the words to one bus, all of the second bits of the words to a second bus, and all of the n^{th} bits of the words to an n^{th} bus. The n busses can then be connected to the n sense amps. The bus generator is developed to do such a task.

4.11.3 Basic Cells

A 10-bit bus, shown in Fig. 4.27, will be at the output of each block.

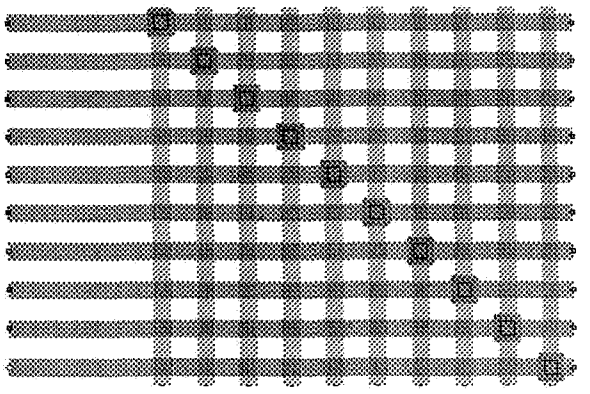


Fig. 4.27 10-bit Bus

The sense amp, shown in Fig. 4.28, has a pre-discharge feature that discharges the sense amp capacitance before it begins sensing. Now when a logic 0 is programmed into memory, the ROM array transistor only needs to pull down the charge on the bit line and not the charge on the bit line and the sense amp. This will speed up the sensing operation. When a logic 1 is programmed into the memory, the output of inverter I1 will go low, turning on the p-transistor which will aid in charging the input of I2, increasing the speed of the sensing operation.

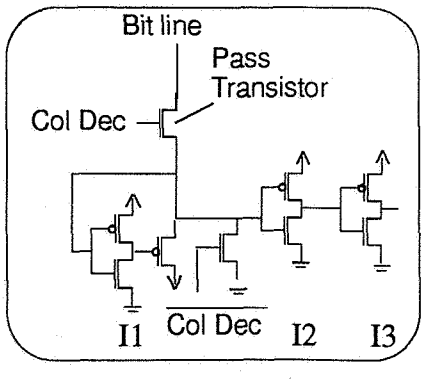


Fig. 4.28 Sense Amp

The column decoder is an array of static AND gates set up as a tree decoder. Each AND gate selects a different block of memory. These are placed in series with the sense amps to save area. A column decoder cell is shown in Fig. 4.29.

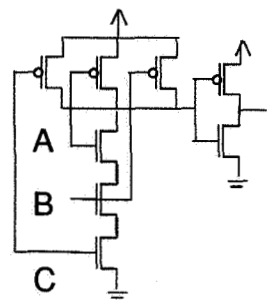


Fig. 4.29 Static Column Decoder

4.11.4 Algorithm

The algorithm for the bus generator is illustrated in Fig. 4.30.

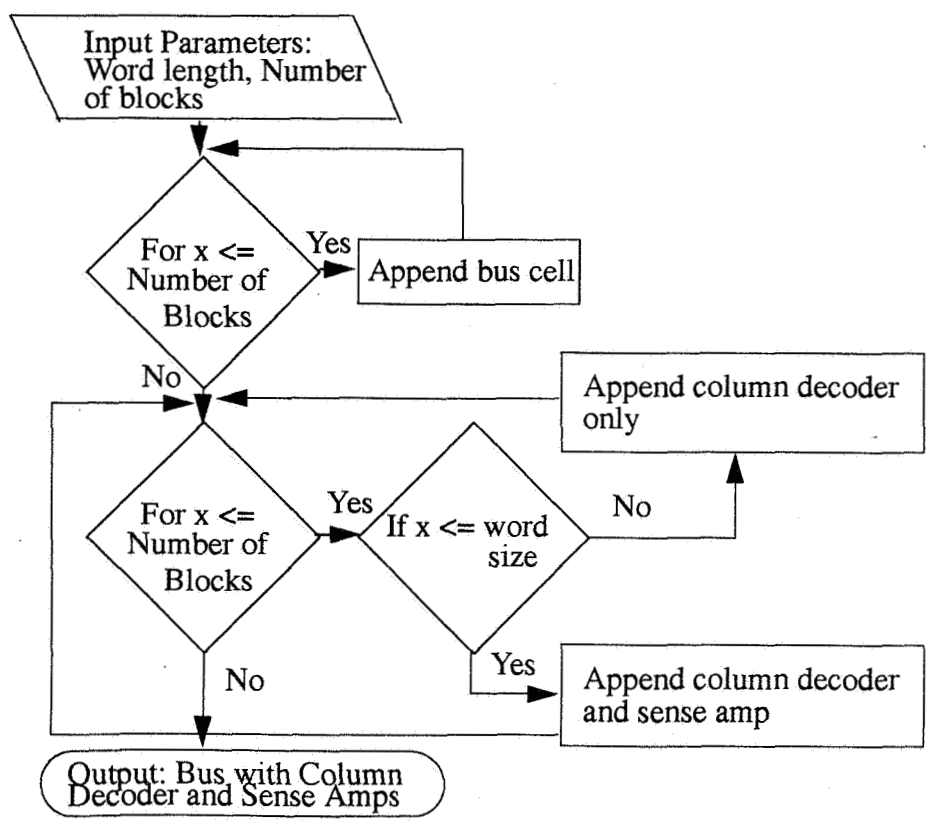


Fig. 4.30 Bus Generator Flow Chart

4.11.5 Output

An example output of the bus generator is shown in Fig. 4.31.

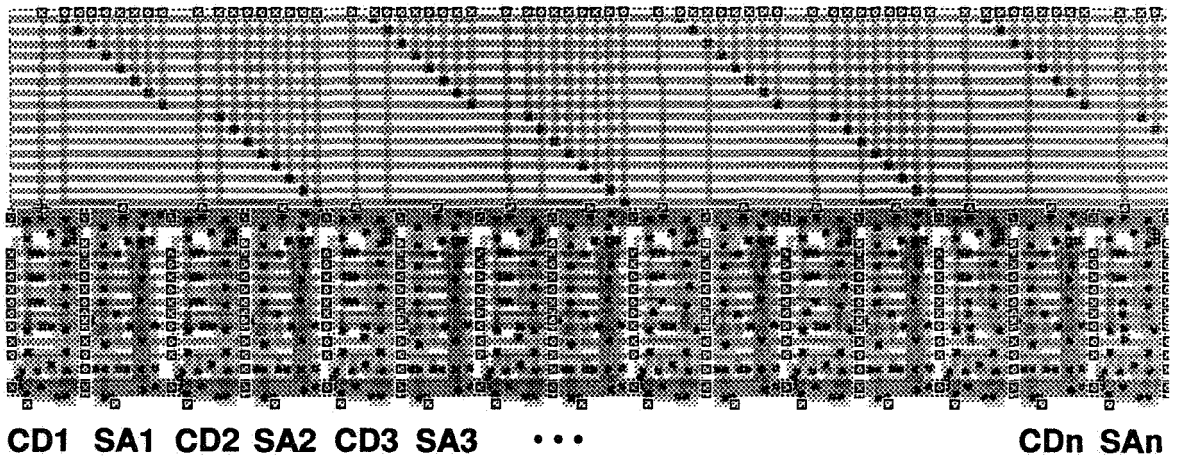


Fig. 4.31 Bus Generator Output

4.12 Decoder Input Buffers

The decoder input buffers in Fig. 4.32 are clocked [12]. Until ϕ goes low, the outputs AA and it's compliment will remain low. This will allow all but one row of the row decoder and column decoder transistors to turn on. This leaves no path to ground during the pre-discharge phase and the outputs will be unaltered. When the pre-discharge phase is over, ϕ goes low and the only transistors that are needed to be turned on are the last stage.

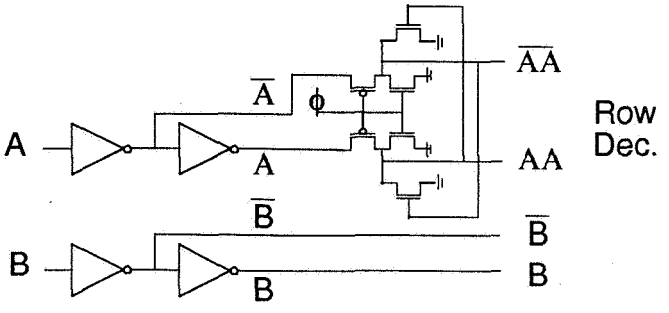


Fig. 4.32 Decoder Buffers

4.13 Serial Memory Generator

A serial access memory (SAM) is used in every component of the demodulator system. It is primarily used as a pipelining unit.

4.13.1 Basic Cells

The serial access memory, shown in Fig. 4.33, is a dynamic register with a few modifications. The input has an AND gate for resetting the circuit. Also, the register requires only one clock signal. This is done by having the first stage latched with an n-transistor and the second stage latched with a p-transistor. Transistor P1 is placed in the circuit in order to increase it's speed. The n-transistor can not pass a true logic 1, so the P1 transistor will sense the output of the inverter. When the output goes below the threshold voltage of P1, it will turn on and bring the input to a true logic 1. Transistor N1 serves a similar purpose. Since the p-transistor can not pass a true logic 0, the N1 transistor senses the output of the second stage inverter and brings the input to a true logic 0.

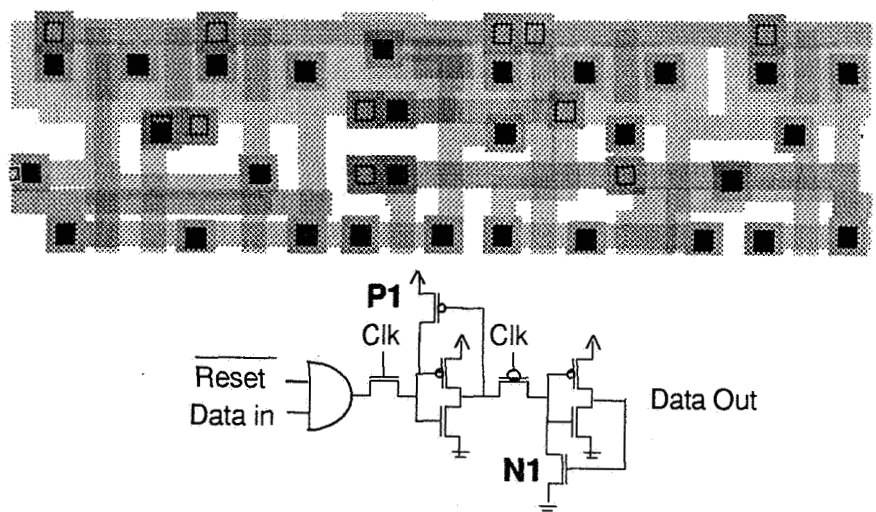


Fig. 4.33 SAM cell

4.13.2 Algorithm

The input parameters for the SAM generator are the number of rows and columns of the SAM needed. This will specify the organization of the cell and the algorithm has to append the basic cells to get such an organization. Fig. 4.34 illustrates the algorithm used in the SAM generator.

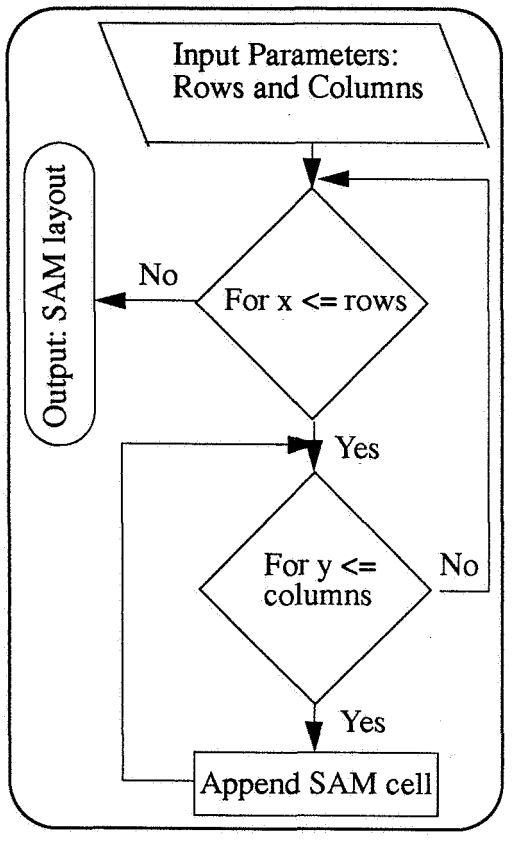


Fig. 4.34 Serial Access Memory Algorithm

4.13.3 Output

The output of a 5x3 SAM is shown in Fig. 4.35.

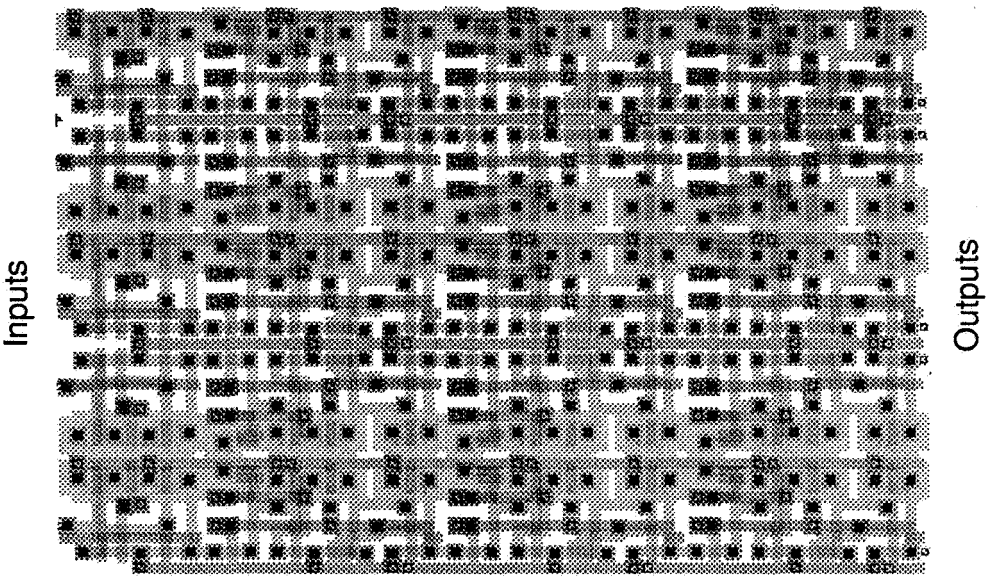


Fig. 4.35 Serial Memory Array (5 x 3)

The simulation for the 5 x 3 SAM is shown in Fig. 4.36. As seen from the simulation, the 5-bit word is delayed three clock cycles.

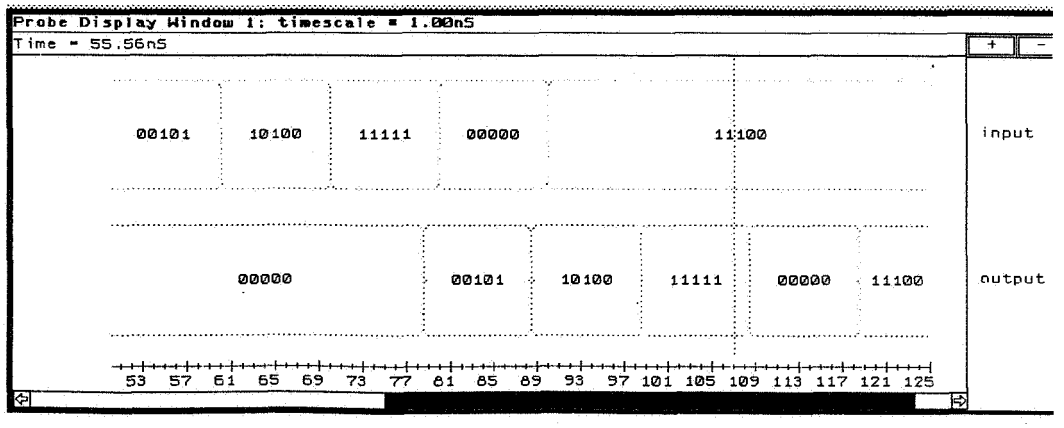


Fig. 4.36 Simulation of 5 x 3 SAM

Chapter 5 Main Components

5.0 Development of Main Components

Now that component generators have been developed for the most commonly used components, it is time to use these to construct the main components. Once the generated components are placed in a satisfactory manner, MicroRoute will be used to do the block routing for the system.

5.1 Numerically Controlled Oscillator

As stated in Chapter 3, the NCO needs to provide samples of $\cos(2\pi f_c nT)$ and $\sin(2\pi f_c nT)$. Therefore, the argument $2\pi f_c T$ needs to be accumulated each clock cycle by a binary accumulator. An accumulator is shown in Fig. 5.0

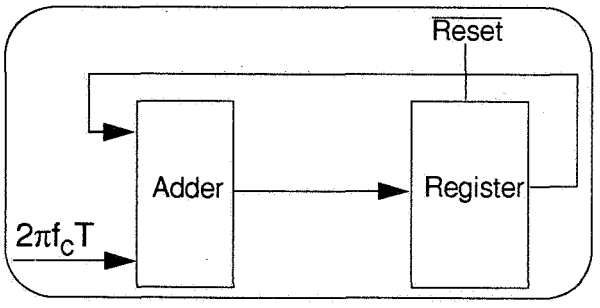


Fig. 5.0 Binary Accumulator

The accumulation of a binary number will result in an overflow as soon as the accumulation becomes larger than can be handled by the adder and the register. In order to avoid the need for circuitry to control this overflow, a technique can be

used to design an NCO that does not care about the overflow. Consider a 10 bit system where the phase being accumulated is represented as a 10 bit binary number. As soon as the accumulation results in an 11 bit number, an overflow has occurred. Assume that the binary representation of 2π is an 11 bit number equal to 10000000000. The fact that any phase subtracted from 2π will result in the same angle will resolve the overflow problem. A simple truncation of the 11^{th} bit is the same as subtracting 2π from the phase. For example, assume that the phase increment is 0100000000 and the last accumulation resulted in the binary number 1111111111. After the next accumulation, the result will be 1001111111 which is approximately equal to 2.5π. If the 11^{th} bit is truncated then the result is approximately 0.5π which is the same angle as 2.5π and is represented by a 10 bit binary number. See Fig. 5.1 for more on this example.

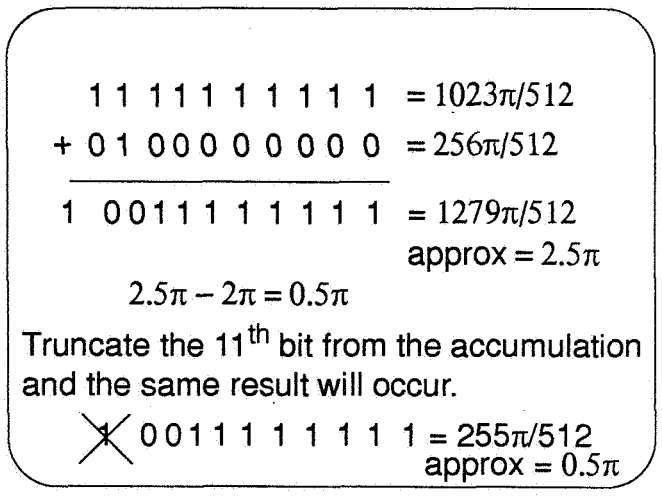


Fig. 5.1 Accumulation Example

The result is a free running NCO that needs no control circuitry other than a simple reset. The phase recovery unit also provides a 10 bit phase estimate. This is done by inserting an adder after the accumulator to add in the phase estimate. This is shown in Fig. 5.2.

The argument $2\pi f_c nT + \theta_e$ is then given as an input to a ROM which is used as a look-up table for the values of $\cos(2\pi f_c nT + \theta_e)$ and $\sin(2\pi f_c nT + \theta_e)$. There are 1024 locations of data which corresponds to a phase accuracy of $\pi/512$. The phase increment needs to be converted to a binary number by $\frac{n\pi}{512} = 2\pi f_c nT$. From this relation, $n = \pi/2$, so the binary representation for the phase increment is 0100000000. This increment can not be changed to any other number since the filters are designed for a fixed sampling frequency. If adaptive filters are used, then there would be some flexibility for the phase increment.

5.1.1 Organization and Layout

Table 3 shows a list of the components that are needed to construct the NCO. The adders and the SAMs are combined to form the accumulators and the ROM is used to store the $\sin(\omega t)$ and $\cos(\omega t)$ look up table data.

Table 3: NCO Components

Component	Size	Number
ROM	2048 x 10 bit	1
Adder	10 bit	2
SAM	10 by 1	2

The organization of the NCO is shown in Fig. 5.2.

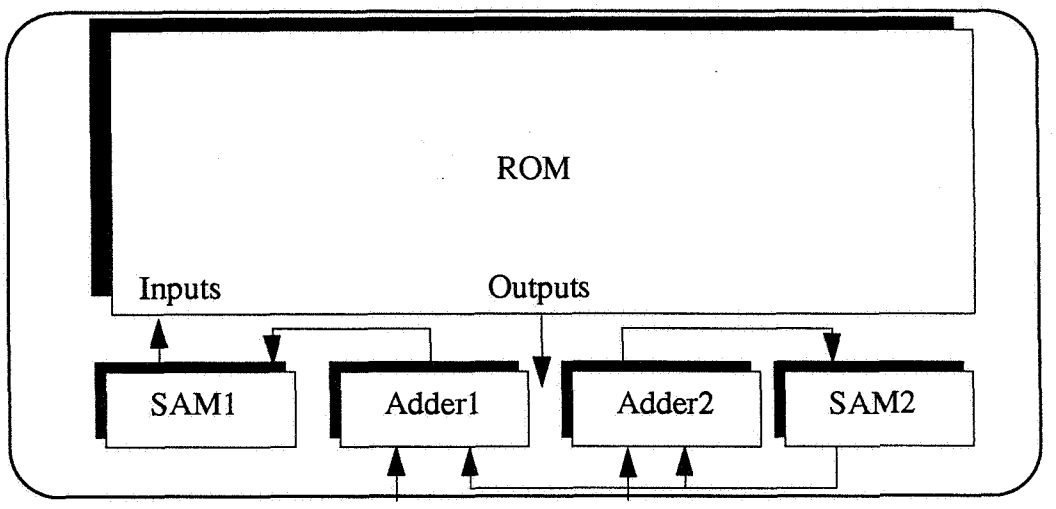


Fig. 5.2 Organization of the NCO

The layout of the NCO is more long than wide for pitch matching reasons. The inputs consist of the two clocks that are used by the ROM and the SAMs, a reset and the 10-bit phase estimate from the PRU. The layout of the NCO is shown in Fig. 5.3.

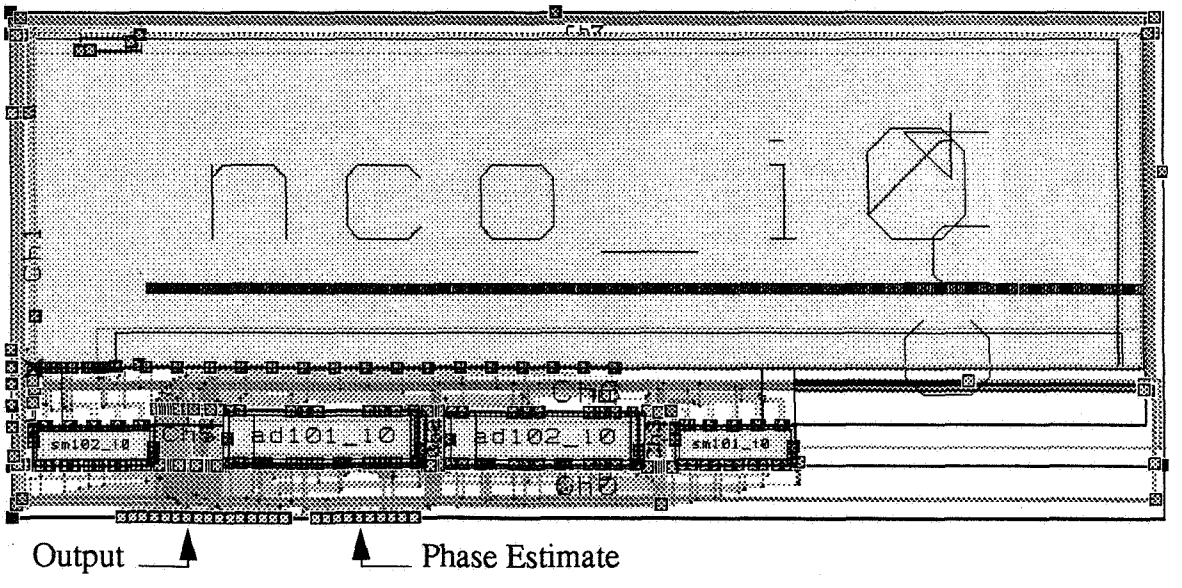


Fig. 5.3 Layout of NCO

5.1.2 Simulation

The simulation of the NCO is shown in Fig. 5.4. The Sin(ωt) and Cos(ωt) outputs repeat themselves after every four samples. This is because this system is designed to receive signals represented by 4 samples / symbol.

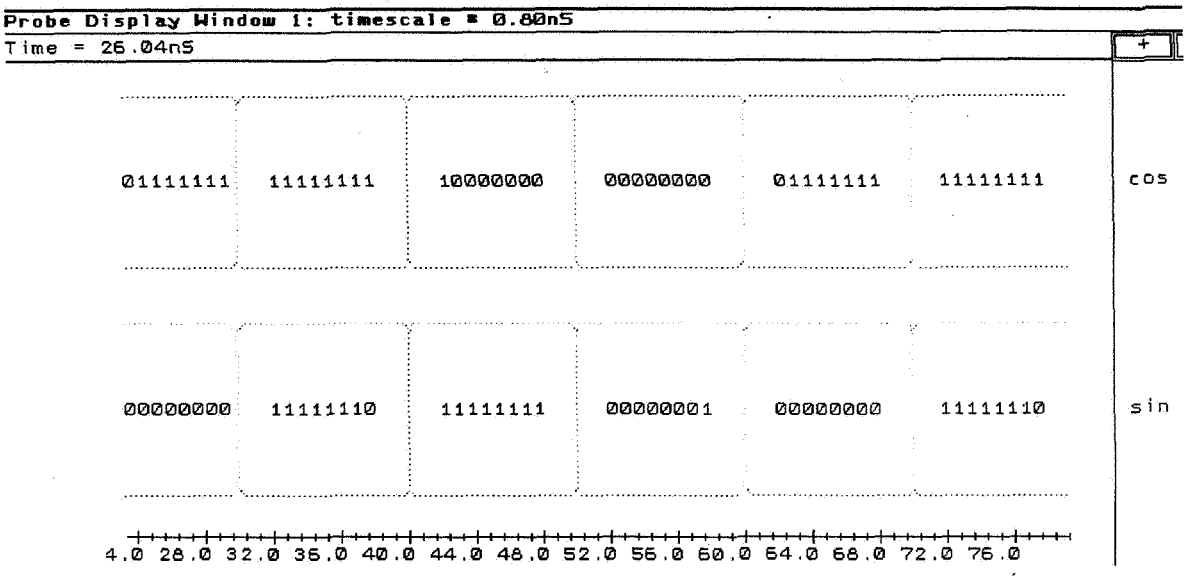


Fig. 5.4 Simulation of the NCO

5.2 Lowpass Filter

The equation for the filter is:

$$H(z) = -0.034\,(z^{7} + z^{-7}) + 0.055\,(z^{5} + z^{-5}) - 0.101\,(z^{3} + z^{-3}) + 0.316\,(z^{1} + z^{-1}) + 0.499$$

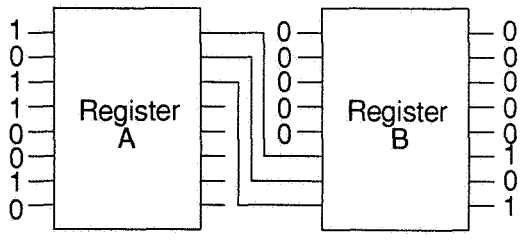
The coefficients are generated from the Parks McClellan algorithm at NASA. It can be seen from the equation that there are five multiplications that are needed to do the filtering operation. Multipliers are very large and take long periods of time to evaluate, so it would be advantageous to not have these. There is a technique that can be applied to digital filters that will require no multipliers to do the multiplication. Multiplication can be done by simply shifting the data to the left or to

the right a certain number of bits. A number multiplied by 2^n, where n is an integer, would be shifted to the left n places if n is greater than zero and shifted to the right n places if n is less than zero. To apply this to digital filtering, the coefficients must be converted to the nearest power of two value. The new coefficients are shown in Table 4.

Table 4: Power of Two Coefficients

Coefficient	2^n Representation
-0.03125	2^{-5}
0.0625	2^{-4}
-0.09375	$2^{-4} + 2^{-5}$
0.3125	$2^{-2} + 2^{-4}$
0.5	2^{-1}

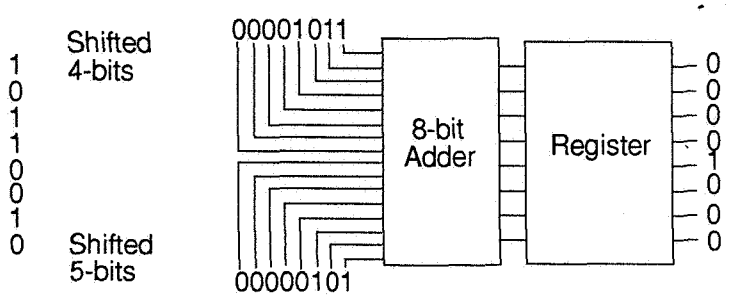
In the case where the coefficient is described by one multiplier such as 2^{-5}, a simple shift in wiring is all that is needed. As an illustration, Fig. 5.5 shows the multiplication of (10110010_2) by 2^{-5}. Register A takes the binary word ($10110010_2 = 178_{10}$) at it's input and holds it at the output. The shift by 5 is hard wired from the output of register A to the input of register B. The output of register B is then the original binary number multiplied by 2^{-5}.



2^{-5} 178 = 5.56 00000101=5

Fig. 5.5 Sample Multiplication

In the case where the coefficient is described by the addition of two numbers, such as $2^{-4} + 2^{-5}$, an adder is needed to complete the multiplication. Fig. 5.6 shows an illustration of the multiplication of (10110010_2) by $(2^{-4} + 2^{-5})$. The 8-bit word is shifted 4 times to the right and added to the same 8-bit word shifted 5 times to the right. The register then holds the result of the multiplication.



$2^{-4}\ 178 + 2^{-5}\ 178 = 16.68 \quad 00001000 = 16$

Fig. 5.6 Sample Multiplication

In both examples the answer is always approximately correct if we truncate what is after the decimal point. There is no loss if the fractional part is not truncated. Of course there will be a need to truncate some of the fractional part in order to keep the word size at a practical length. The trade off is a loss in accuracy for the reduction of area and an increase in speed. Obviously the area and speed saved here more than compensate for the loss of accuracy, so the filter should be implemented with no multipliers.

5.2.1 Organization and Layout

Table 5 shows a list of the components that are needed to construct the lowpass filter. This table makes it clear why the component generators are needed.

Table 5: LPF Components

Component	Size	Number	Component	Size	Number
Adder	10 bit	6	SAM	8 by 1 bit	2
Adder	8 bit	2	SAM	8 by 1 bit	2
Adder	6 bit	2	SAM	8 by 3 bit	1
SAM	7 by 4 bit	1	SAM	6 by 2 bit	1
SAM	10 by 1 bit	1	SAM	9 by 1 bit	1
SAM	7 by 1 bit	1	SAM	6 by 1 bit	1
SAM	5 by 1 bit	1	SAM	4 by 1 bit	1
Buffers		6			

The organization of the filter is shown in Fig. 5.7 and the layout is shown in Fig. 5.8. The inputs are a clock input, a reset and an 8-bit data word. The output is only an 8-bit word.

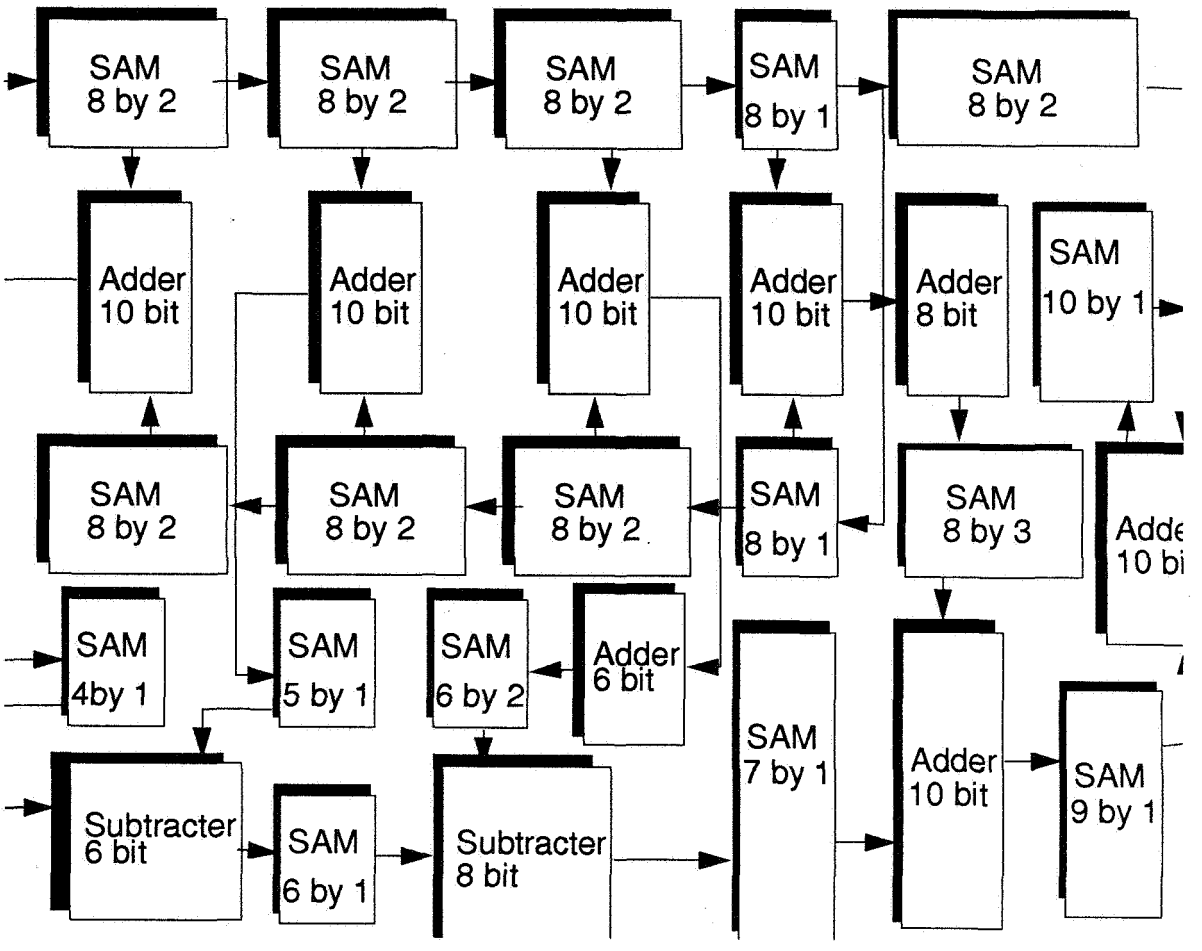


Fig. 5.7 Organization of Lowpass Filter

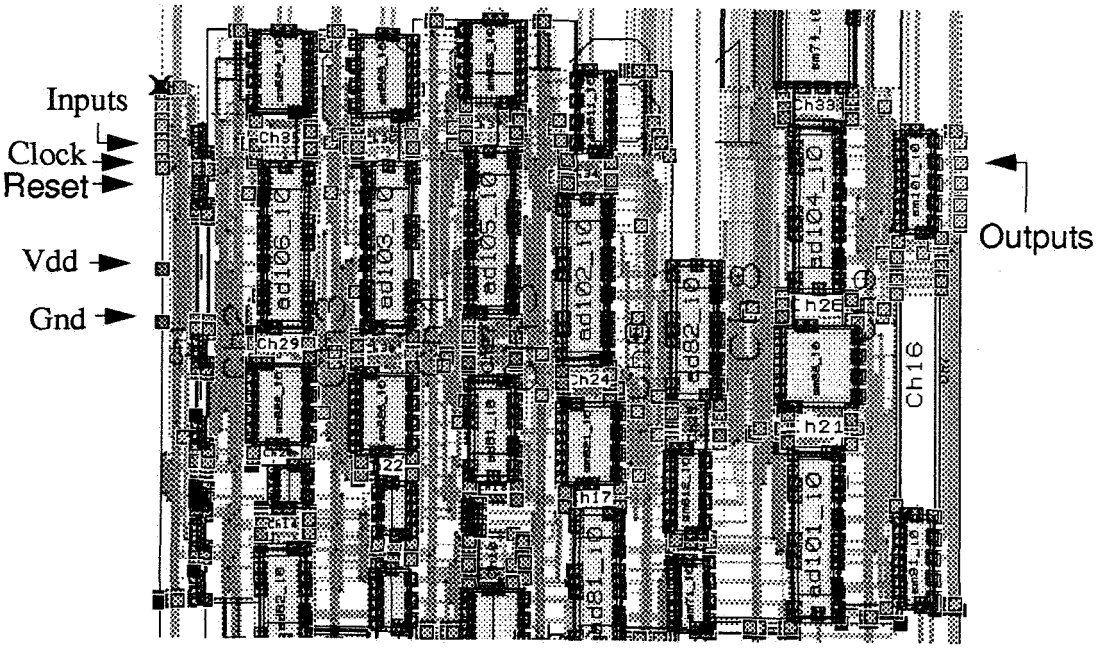


Fig. 5.8 Layout of Lowpass Filter

5.2.2 Simulation

Fig. 5.9 shows how the amplitude response of the Parks/McClellan lowpass filter. The spectrum of the signal after the filter is shown in Fig. 5.10. Note that these two responses look very similar.

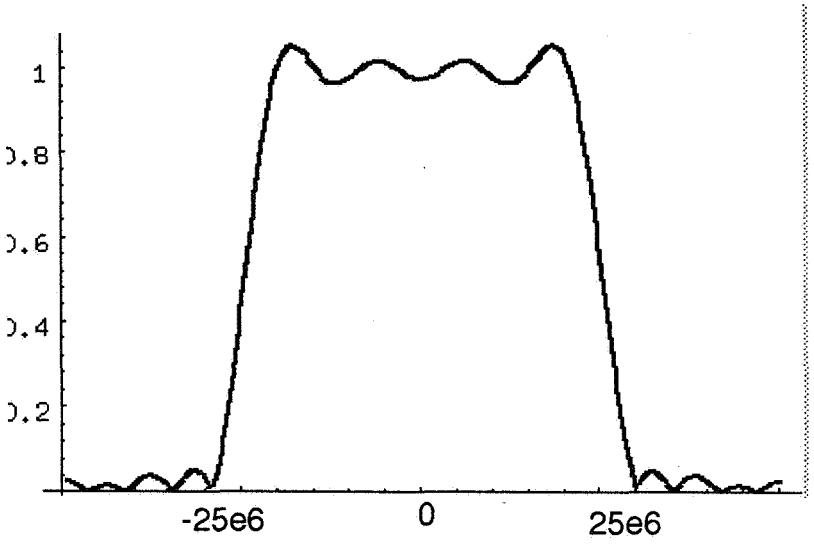


Fig. 5.9 Amplitude Response of the Lowpass Filter

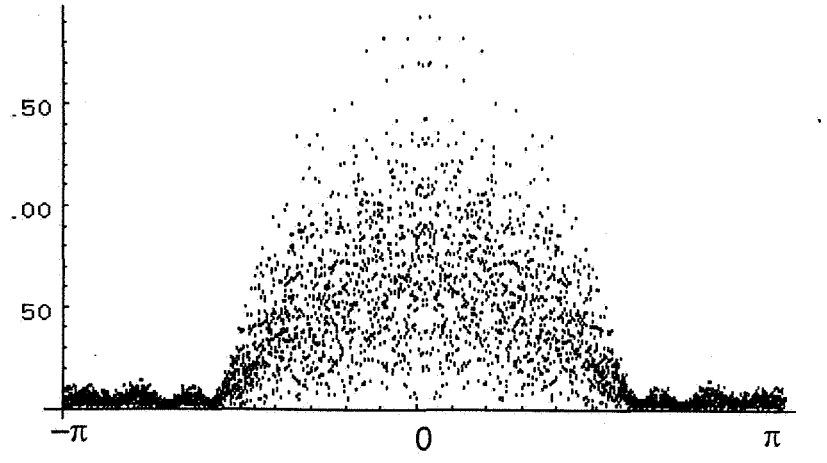


Fig. 5.10 Spectrum After Filter

5.3 Integrate and Dump Unit

The IDU accumulates 4 clock cycles of data and makes a decision of which bit is sent. If a logic 1 is sent, then the data is expected to be positive. If a logic zero is sent, then the data is expected to be negative. In two's complement mathematics, the sign bit is the most significant bit. If the most significant bit is a logic 1, then the data is said to be negative and if the most significant bit is a logic 0, then the data is positive. After an accumulation of positive data, the sign bit will be logic 0 and after an accumulation of negative data, the sign bit will be logic 1. Therefore, the sign bit needs to be inverted in order to receive the correct data.

The IDU is made up of an accumulator, and a latch with an inverted output. The organization of the IDU can be seen in Fig. 5.11.

5.3.1 Layout and Organization

All of the components used for the IDU are shown in Table 6.

Table 6: Component list for IDU

Component	Size	Number
Adder	10 bit	1
SAM	10 bit	1
SAM	1 bit	1
Counter	4 state	1

The layout of the IDU is shown in Fig. 5.12. The layout is very small and simple.

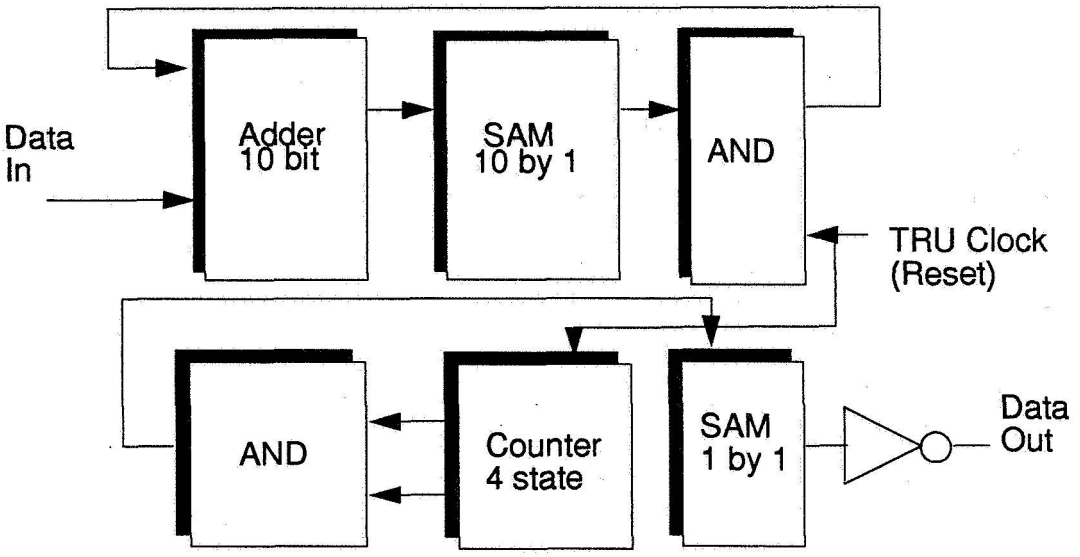


Fig. 5.11 Organization of Integrate and Dump Unit

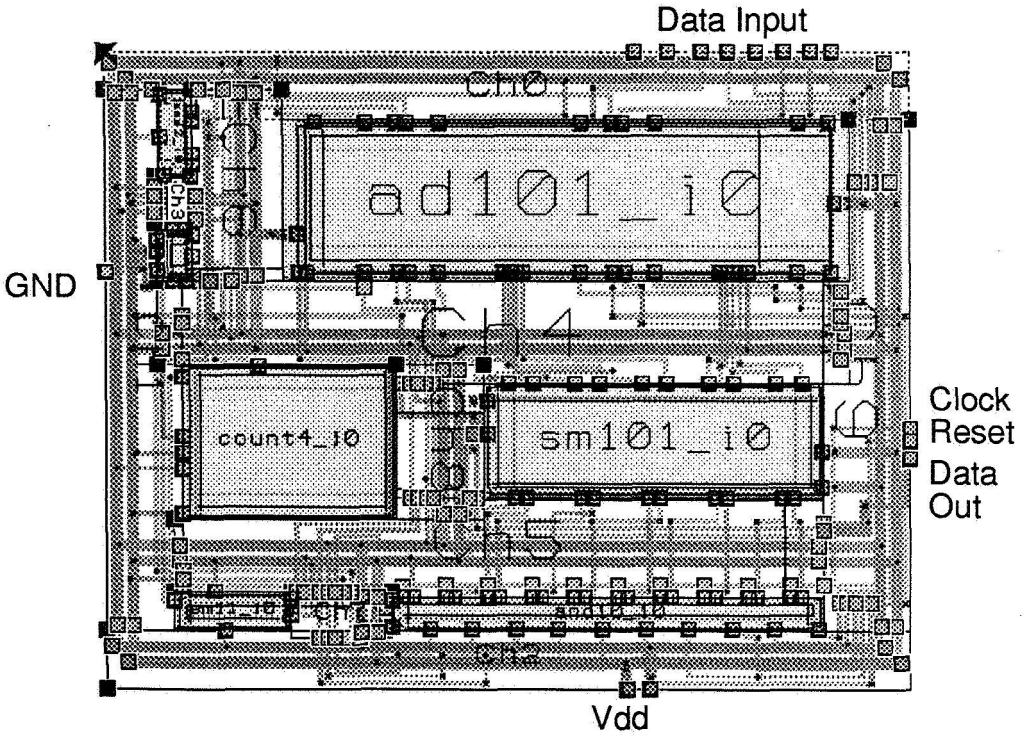


Fig. 5.12 Layout of IDU

5.3.2 Simulation

The simulation of the IDU is shown in Fig. 5.13. After four

accumulations, the output will be the inverse of the sign bit.

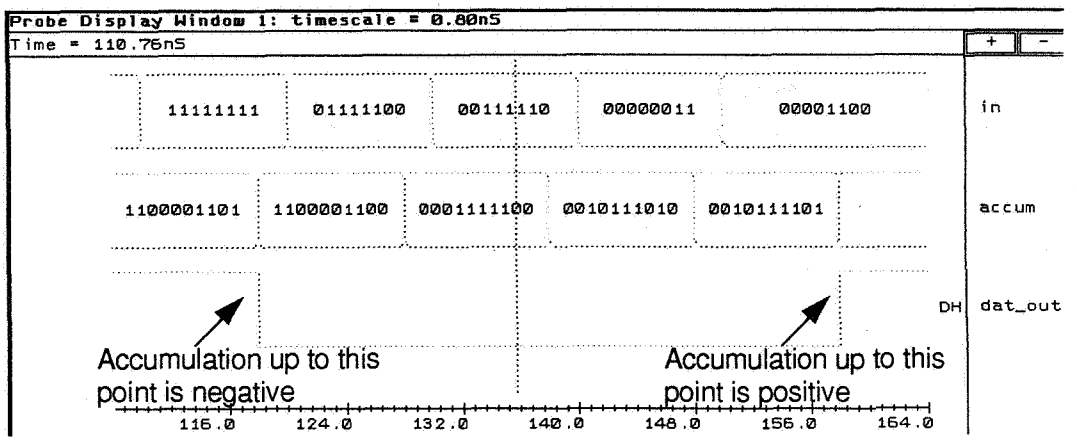


Fig. 5.13 Simulation

5.4 Phase Recovery Unit

The PRU requires that there be two accumulators in its architecture. The accumulators are designed as in Fig. 5.14. One will accumulate 4 samples and the other will be designed to accumulate 8 symbols.

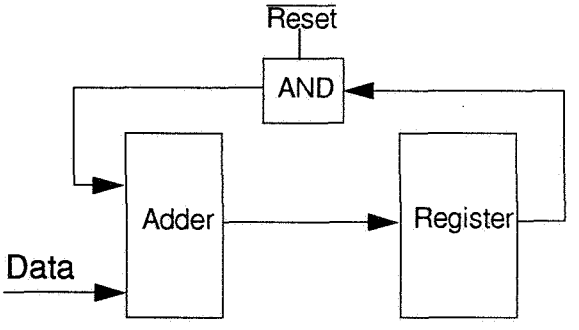


Fig. 5.14 Accumulator Structure

The phase recovery unit needs different clocks for proper operation. For example, the PRU has a 4-bit accumulator that outputs it's sum after 4 clock cycles and an 8-bit accumulator that outputs it's sum after 8 clock cycles. Different clocks must be used to enable the outputs of these accumulators at different times.

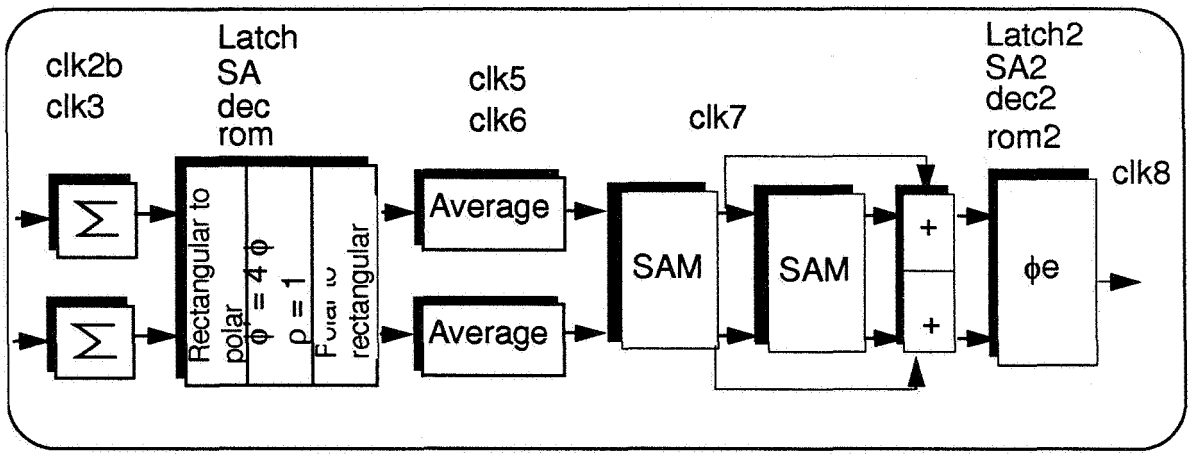


Fig. 5.15 PRU Clocking Scheme

It can be seen from Fig. 5.15 that the clocking scheme for the PRU is very messy and will require careful design. The PRU clock is shown in Fig. 5.16.

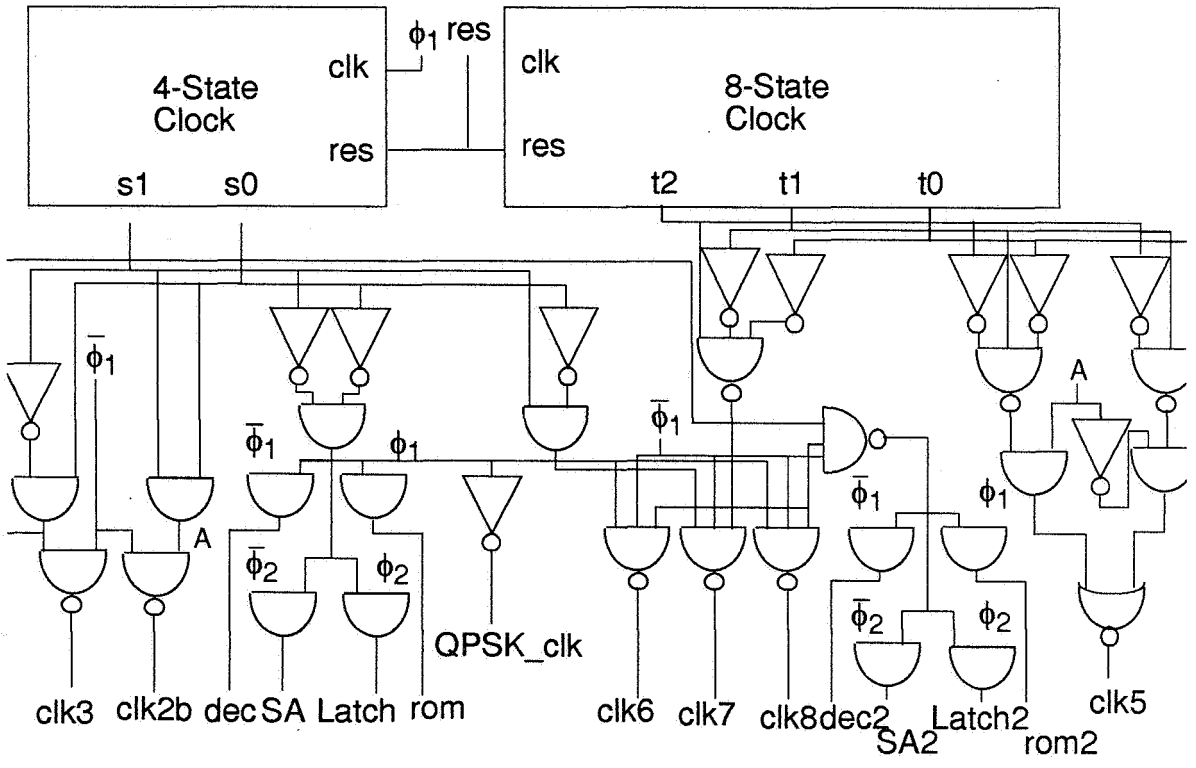


Fig. 5.16 PRU Clock Unit

The 4 sample accumulators use clk2b as a reset and clk3 as the main clock. The ROMs use Latch and Latch2 as clocks for the latches, dec and

dec2 as clocks for the column decoders, SA and SA2 as clocks for the sense amps and rom and rom2 for the pre-charge clocks in the ROM arrays. The 8 sample accumulator uses clk6 as it's main clock and clk5 as a reset. Clk7 drives the two SAMs that hold the N symbols before and N symbols after the symbol whose phase is being estimated. Clk8 is used to latch the output of the phase ROM.

All of the clocks here are carefully buffered such that all of the signals propagate when they are supposed to. The clocks are made with CMOS D type flip-flops. A reset is incorporated into the counters so that when the system resets, so would the counters. The 4-state counter is shown in Fig. 5.17.

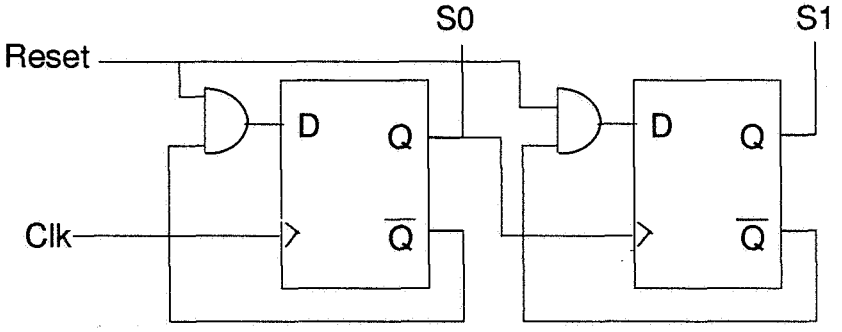


Fig. 5.17 4-State Counter Circuit

5.4.1 Layout and Organization

Table 7 shows a list of the components to be used for the PRU.

Table 7: Components List for PRU

Component	Size	Number	Component	Size	Number
Adder	10 bit	2	Adder	8 bit	4
SAM	6 by 1	10	SAM	10 by 1	1
SAM	9 by 1	2	SAM	8 by 1	2
ROM	4096 by 12	1	ROM	4096 by 10	1
PRU Clocks		1			

The organization of the PRU is shown in Fig. 5.18 and the layout is shown in Fig. 5.19.

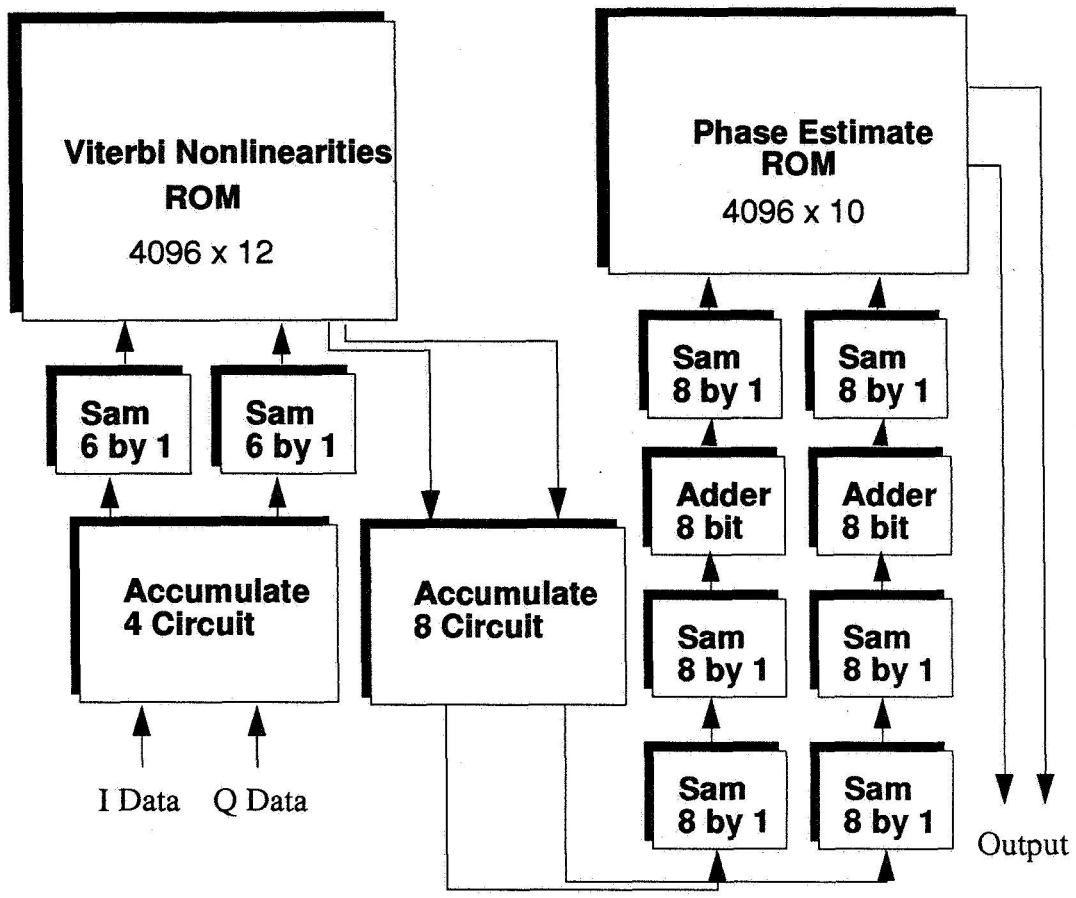


Fig. 5.18 Organization of Phase Recovery Unit

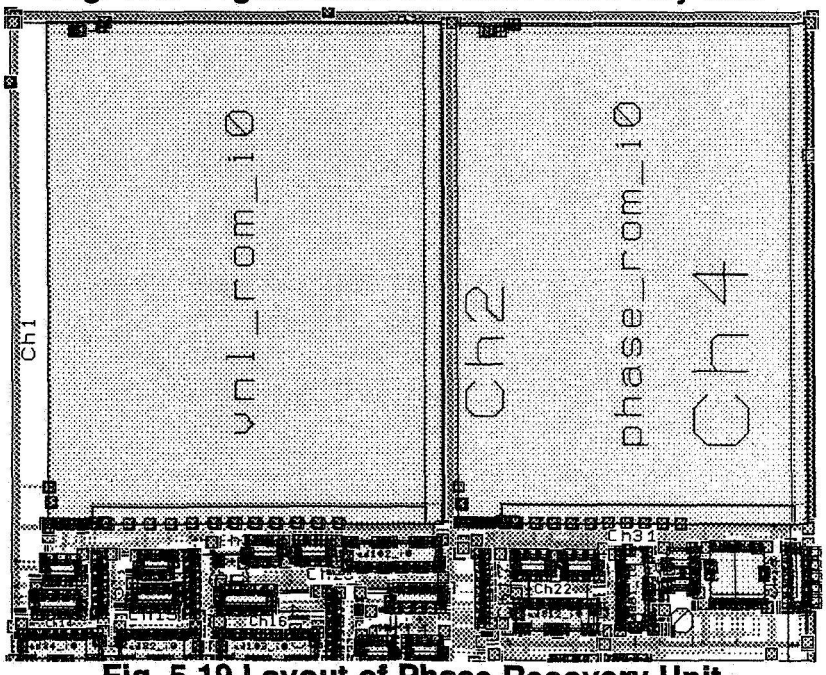


Fig. 5.19 Layout of Phase Recovery Unit

5.4.2 Simulation

A portion of the simulation for the PRU is shown in Fig. 5.20. A zero phase error signal is introduced to the demodulator. From the simulation, the detector is estimating the phase to be approximately $2\pi = 0\pi$ which is approximately zero phase error.

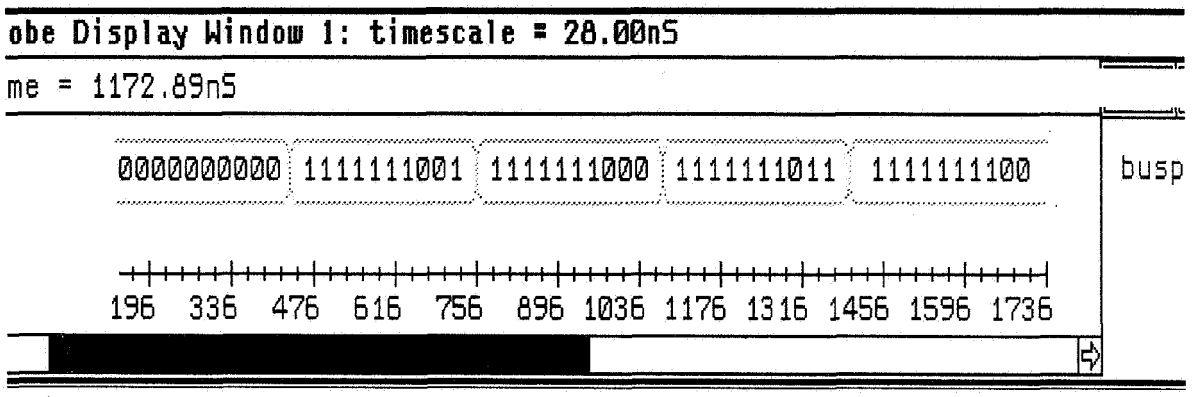


Fig. 5.20 Simulation of the PRU

5.5 Timing Recovery Unit

The TRU is used to extract the timing from the signal. It makes a decision based on the number of consecutive zero crossings. If there are three consecutive positive samples paired with three consecutive negative samples, or vice versa, then the transition signal will be logic 1. After 3 consecutive transition signals, the symbol clock will be reset.

5.5.1 Layout and Organization

The components used for the TRU are shown in Table 8. The zero crossing detector is made up of the logic gates as shown in Fig. 3.8, and the 4 state counter is made from D type flip-flops.

Table 8: Components List for TRU

Components	Size	Number
Zero Crossing Detector		1
SAM	5 x 1 bit	1
SAM	2 x 1 bit	1
SAM	4 x 1 bit	2
Counter	4 state	1
Glue Logic		3

The organization is shown in Fig. 5.21 and the actual layout is shown in Fig. 5.22. There is a clock input and a reset input. The sign bit of the lowpass filtered data is also input to the TRU. The output is the symbol clock.

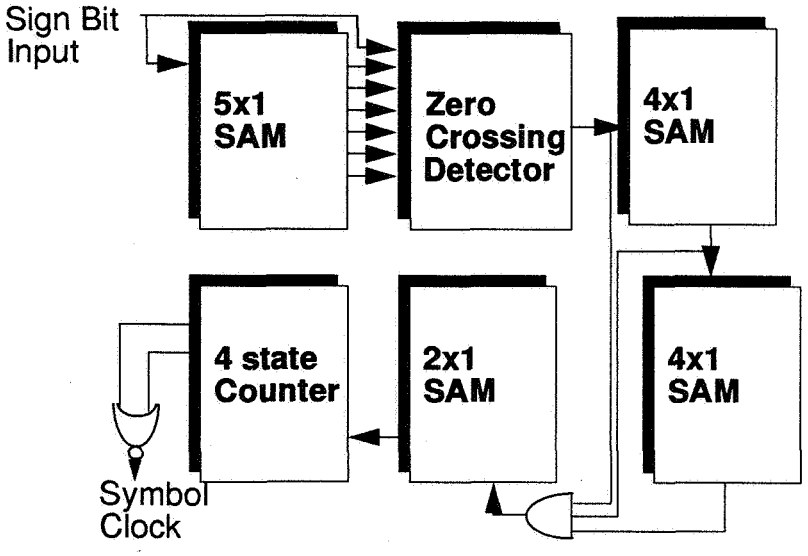


Fig. 5.21 Organization of the TRU

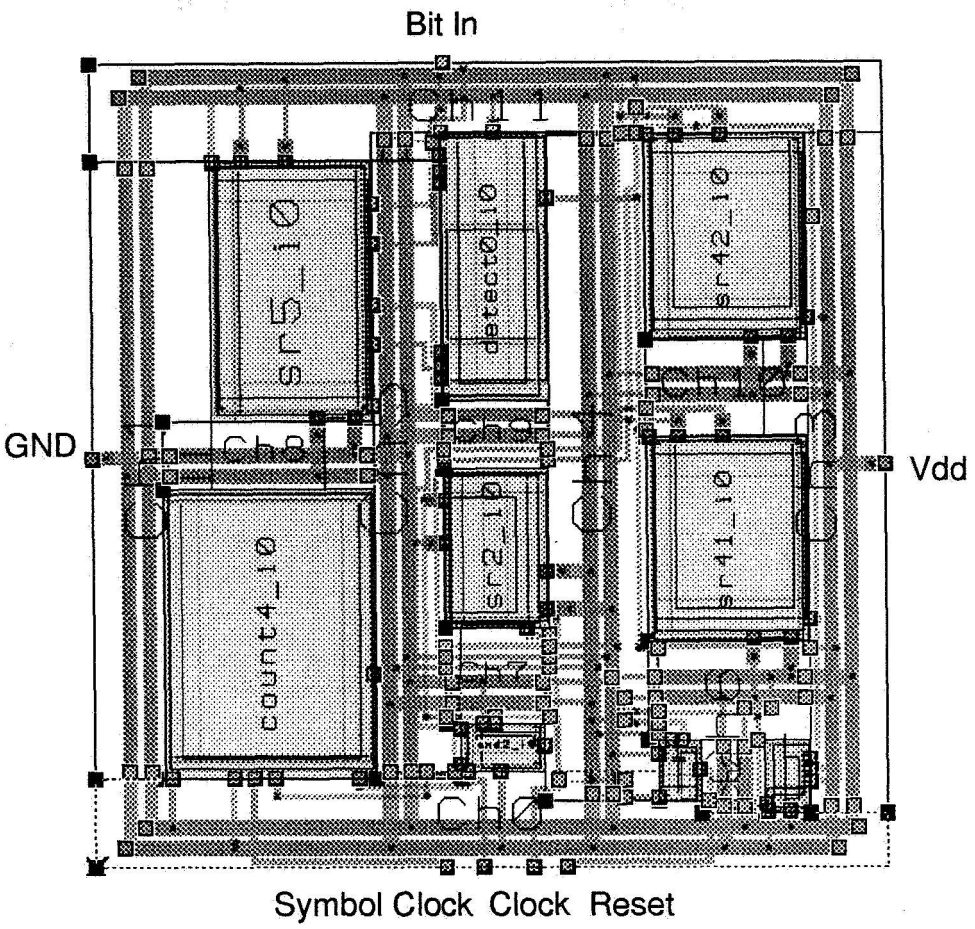


Fig. 5.22 Layout of the TRU

5.5.2 Simulation

The simulation of the TRU is shown in Fig. 5.23. It shows the TRU locking onto the timing of the signal.

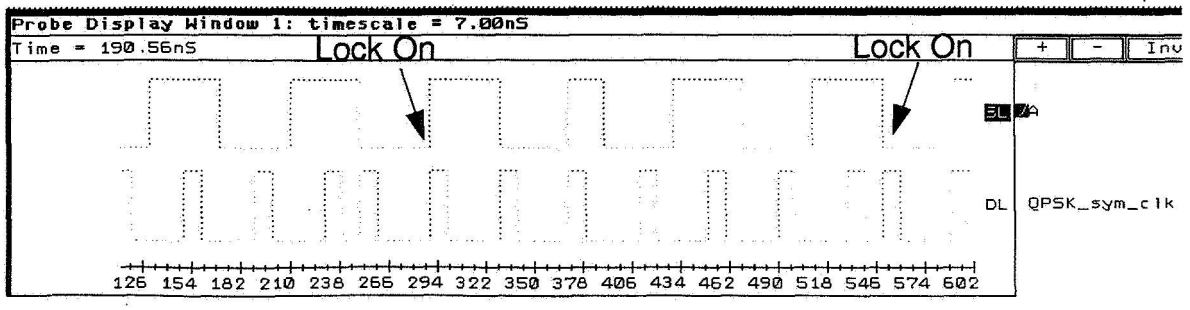


Fig. 5.23 Simulation of Timing Recovery Unit

5.6 Unique Word Detection

A unique word composed of 15 symbols is used to identify the start of the data and to resolve the phase ambiguity. The bits from the IDU are fed into the UWD every symbol period and are compared to the UW stored on the chip. If the UW bits match those in the shift register or are the inverse of those in the shift register, then the data has started. A 15 bit SAM is used to act as a shift register. As an example, consider a 6 symbol unique word UW = 101010. All of the bits in the SAM are initially zero so the output would have 3 places that are incorrect, as shown in Fig. 5.24.

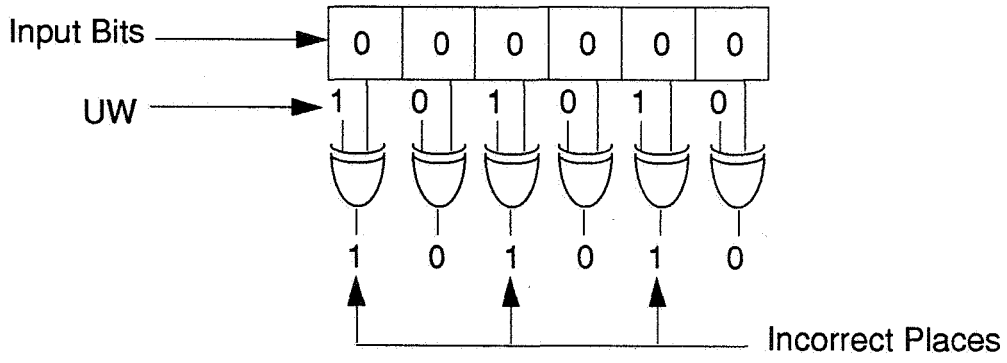


Fig. 5.24 Example of Unique Word Detector

If after some time, the UW appeared in the shift register, then the output would have zero places that are incorrect, as shown in Fig. 5.25.

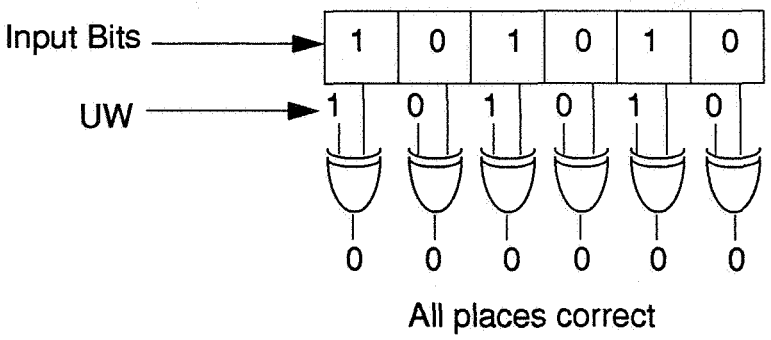


Fig. 5.25 Example of Unique Word Detector

From these examples, it is seen that the number of logic 1 outputs corresponds to the number of incorrect places. So there is a need for a ones counter to give the number of places that are incorrect. An adder cell can be used as a 1's counter as shown in Fig. 5.11. At the output of each of the adder cells is a two bit word that tells how many 1's are at the input. Fig. 5.26 shows that these 2 bit words can be added up to tell the total number of logic 1 inputs.

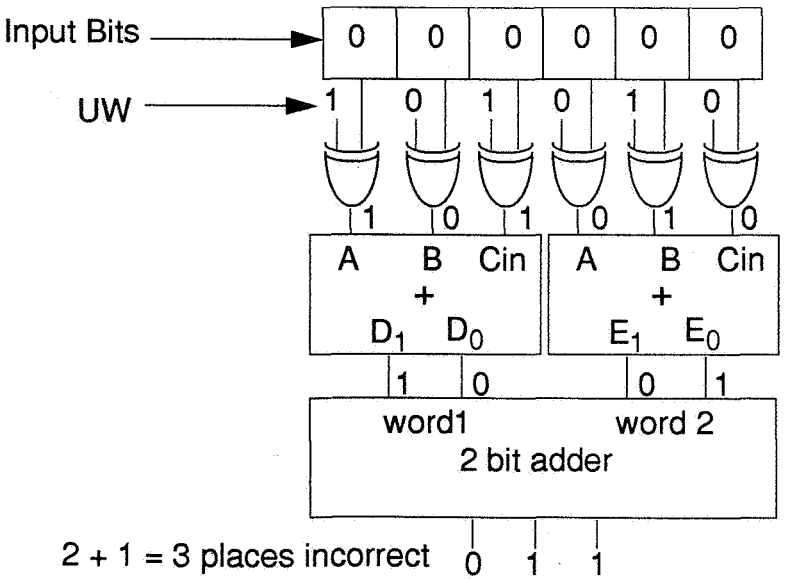


Fig. 5.26 Unique Word Detector

Fig. 5.27 shows a typical output of the UWD.

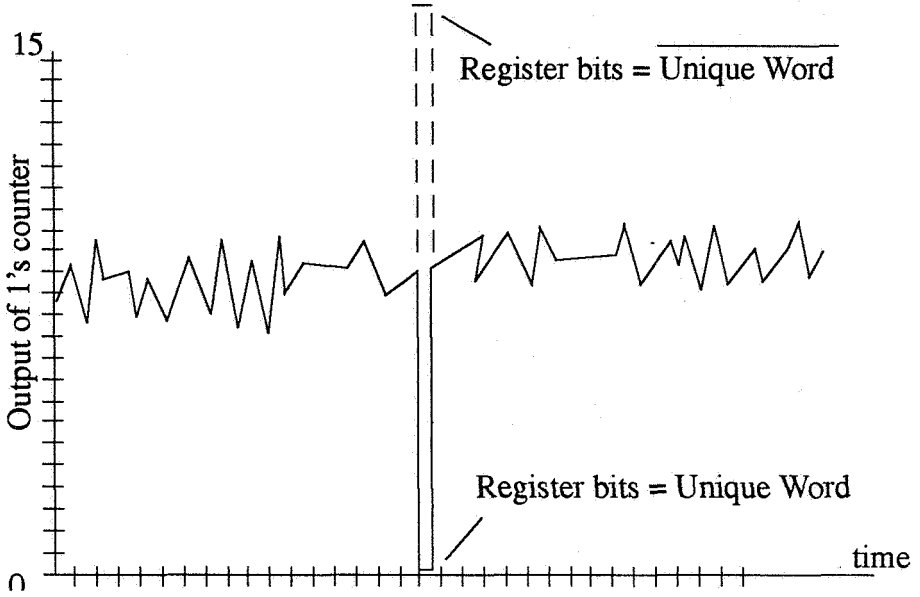


Fig. 5.27 Typical Output of UWD

If the UW is not detected then it is said to be a miss.There is a probability of miss. Because of noise, the UWD should be designed to tolerate a few incorrect bits. The probability of a miss is

$$P_{miss} = \sum_{I=0}^{E} \binom{N}{I} p^{I} (1-p)^{N-I} \qquad (5.0)$$

where E is the number of bits that can be incorrect, I is the number of bits that are incorrect, N is the number of bits in the UW and p is the bit error probability. For a given bit error probability, a probability of miss can be found. For example, in this system, N = 15 and E = 1. If the demodulator performed with $p = 10^{-3}$ then the probability of miss would be $P_{miss} = 10^{-8}$

5.6.1 Organization and Layout

Table 9 shows the component list for the UWD.

Table 9: Component List for the UWD

Component	Size	Number
SAM	15 x 1 bit	1
Adder	1 bit	5
Adder	2 bit	2
Adder	3 bit	2
Adder	4 bit	1

The organization of the UWD is shown in Fig. 5.28 and the layout is shown in Fig. 5.29.

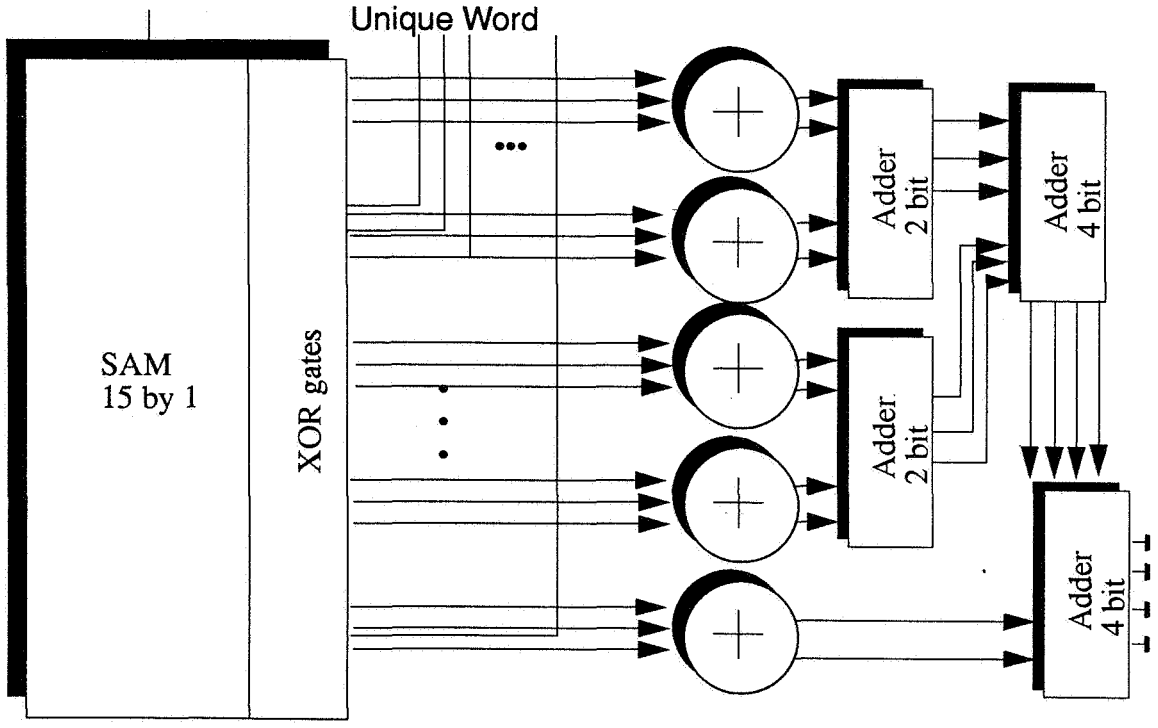


Fig. 5.28 Organization of UWD

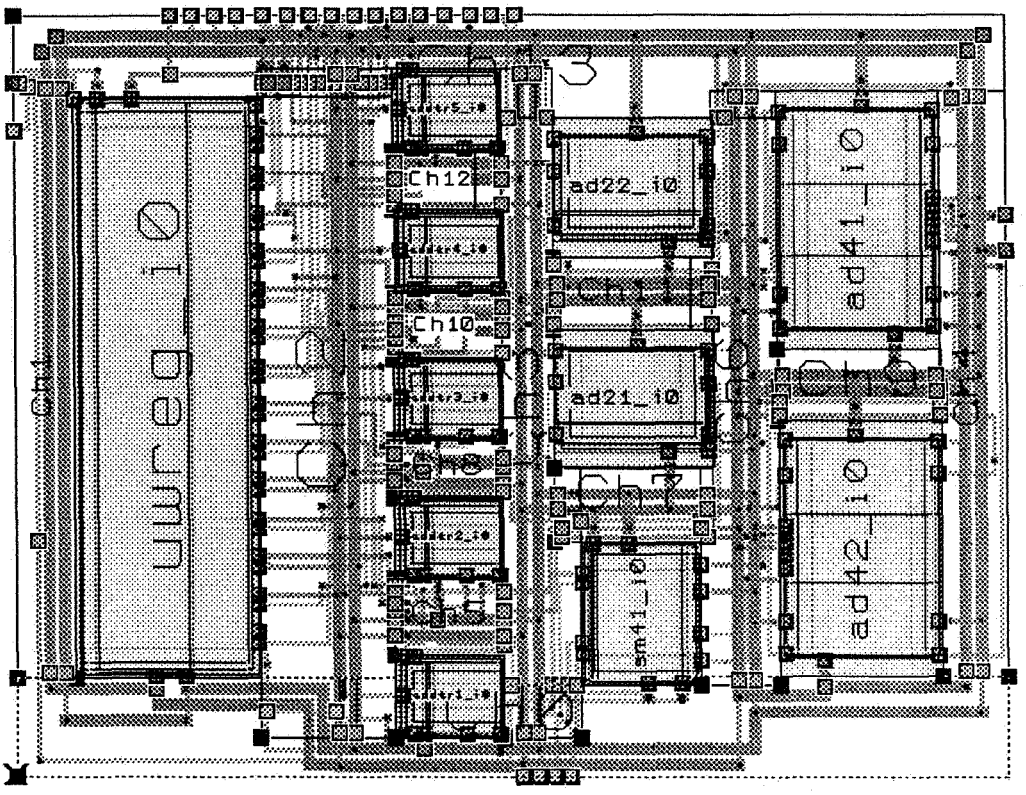


Fig. 5.29 Layout

5.6.2 Simulation

The simulation of the UWD is illustrated in Fig. 5.30. It can be seen from the simulation that when the unique word appears in the registers, the output goes to zero as expected.

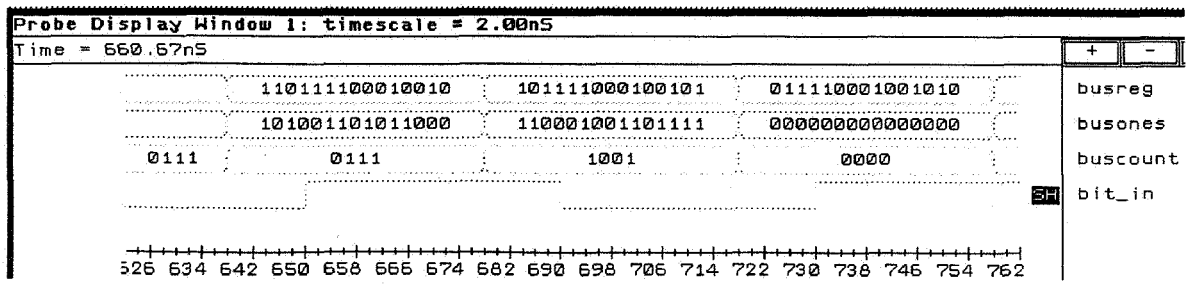


Fig. 5.30 Simulation

Chapter 6 Chip Layout

6.0 Clocking Scheme

Three clocks are needed for proper operation of the demodulator. These are shown in Fig. 6.0.

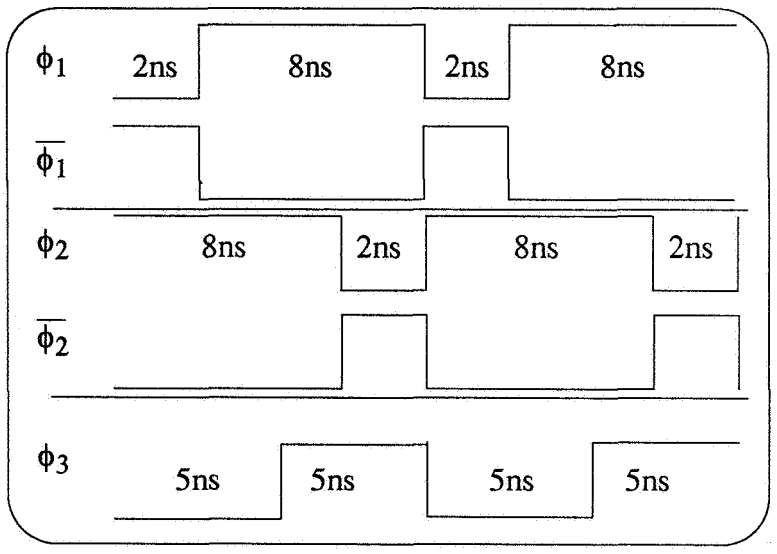


Fig. 6.0 Chip Clocks

In order to keep the complexity minimized, only one 100MHz clock should be used at the input of the chip. This clock is denoted as ϕ_3. Therefore, there is a need for clock shaping on the chip. The circuits shown in Fig. 6.1 are used to do the clock shaping.

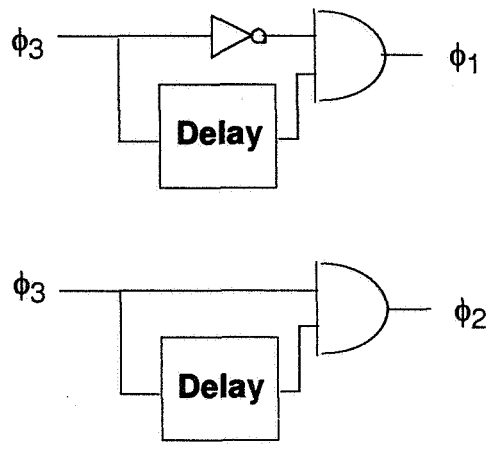


Fig. 6.1 Clock Shaping Circuitry

The delay unit is an RC network made with transmission gates. The resistance and the capacitance can be set by properly sizing the transistor of the transmission gates. So the delay can then be set to whatever is needed by the designer. Fig. 6.2 shows the simulation of the clock shaping circuits.

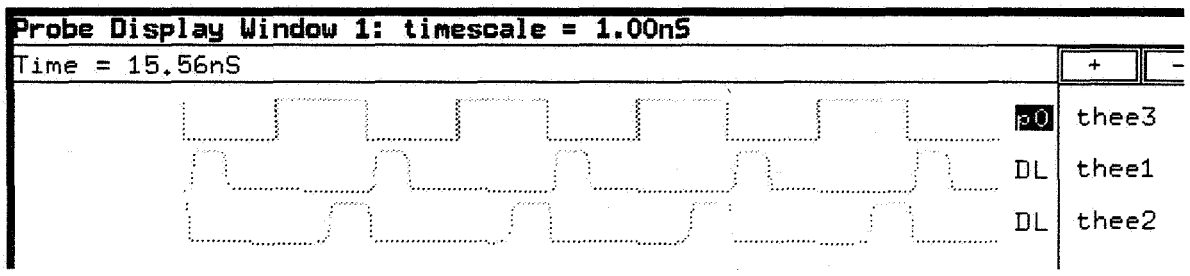


Fig. 6.2 Simulation of Clock Shaping Circuits

The clocks are distributed about the chip in a tree fashion. This will distribute the clocks with equal delays.

6.1 Buffering Scheme

A tree buffering scheme is adopted for this chip design. This is shown in Fig. 6.3. This scheme equalizes the clock delay to each of the components.

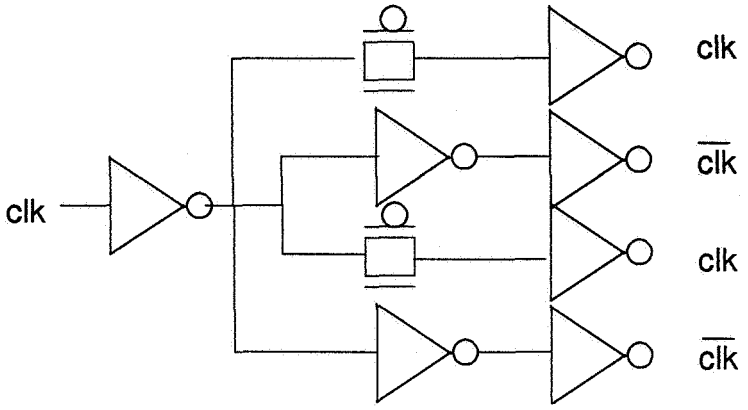


Fig. 6.3 Buffering Scheme

6.2 Arrangement of Components

The components are arranged as shown in Fig. 6.4.

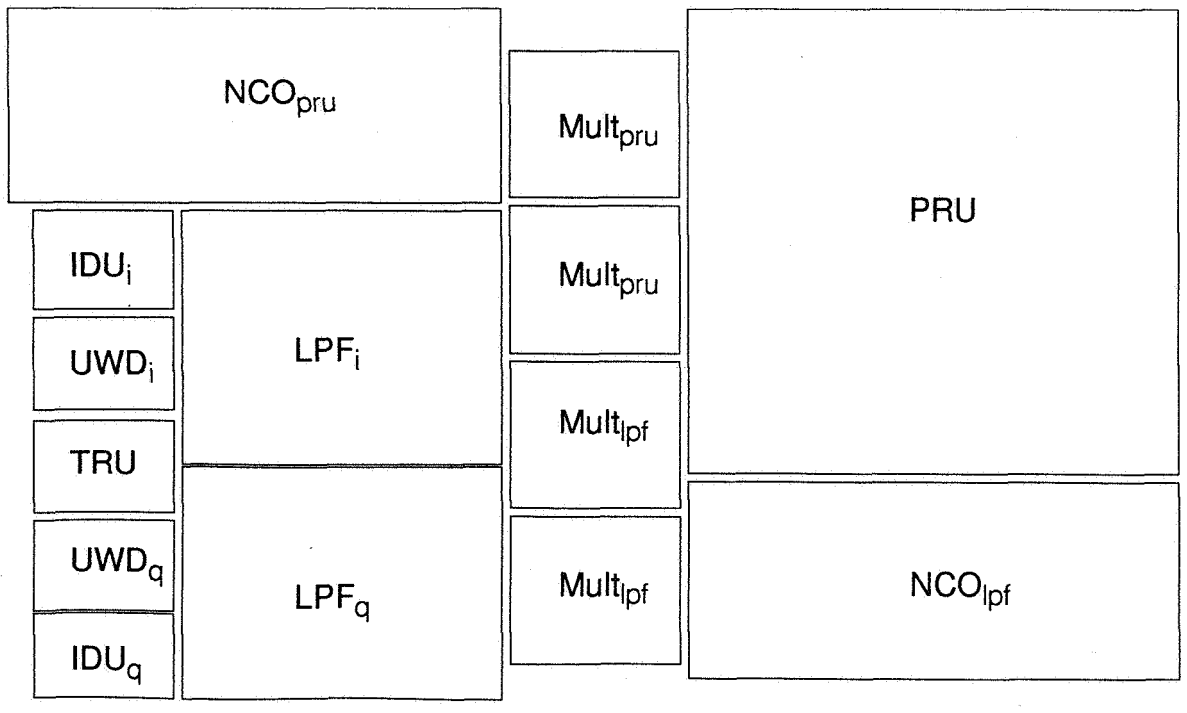


Fig. 6.4 Organization of the Demodulator Chip

6.3 Final Design of Demod

The areas of all of the main components are shown in Table 10.

Table 10: Area of the Main Components

Component	Area (μm^2)
IDU	208 x 262
UWD	310 x 243
TRU	190 x 194
Multiplier	349 x 281
LPF	865 x 770
NCO	1061 x 468
PRU	1507 x 1267

The area of the demodulator is 2.0 x 2.6 mm^2. The total number of transistors is 97,184. The percentage wiring area used is approximately 30%. The demodulator is successfully placed on a single chip.

Chapter 7 Simulation

7.0 Simulation

Simulation plays an important role during all phases of the design and engineering of any communication system [14]. It helps debug any problems that the system may have and it can provide good estimate as to how the system will perform in reality. Simulation of a communication system can be performed in one of two ways: bandpass simulation or baseband simulation. In baseband simulation, there is no need to do any signal upconversions and downconversions since the entire simulation is done at baseband. This type of simulation simplifies the different models in the system and shortens the simulation times considerably. The bandpass simulation performed in this research illustrates the upconversion and downconversion of the transmitted signal. It is a computational burden due to all of the samples that must be taken in order to describe the signal correctly, but it needs to be done to simulate all of the components in the demodulator.

Three forms of system simulation are conducted during the course of this research. A formula based simulation written in the C language is used for initial performance checks. A register based simulation in VHDL is used to represent the demodulator more accurately than the formula based simulation. Layout simulation is done on all of the layout components to verify their functionality and timing. These three simulations will be described in full detail below.

7.1 Formula Based System Representation

The formula based simulation is the first step in creating a communication system. Theoretical equations are used to represent each component and provide the designer with an initial estimate of the system performance. A block diagram of the formula-based system is shown in Fig. 7.0.

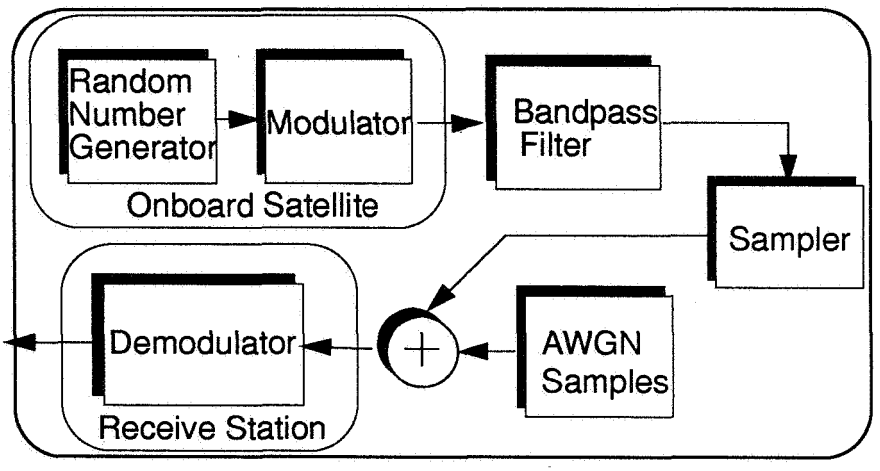


Fig. 7.0 Formula Based System Block Diagram

The assumption that is made here is that the signal is coming from a satellite that has on-board processing. Each of the blocks will be discussed in the following sections.

7.1.1 Pseudo Random Number Generator

Good random number generators (RNG) are very important for simulation. They will produce random symbols to be modulated and random noise to be added to the signal. Unfortunately, all RNG sequences will repeat after a period of time and will produce no new information to the simulation. Therefore it is very important to choose a RNG with a very long period.

The uniform RNG used in this research is found in [13]. This RNG is designed specifically for very long simulation sequences. It combines two different random sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods.

The random number sequences can be generated using the congruential equation:

$$I_{j+1} = (a\, I_j) \bmod (m) \tag{7.0}$$

where j = 0, 1, . . ., a is an integer multiplier, and m is the modulus. The integers a and m are chosen very carefully in order to provide a proper random sequence. The two sequences used in the RNG in [13] are defined by the parameters:

$m_1 = 2147483563$, $a_1 = 40014$

and

$m_2 = 2147483399$, $a_2 = 40692$.

The combination of these two sequences provides a period of about 2.3×10^8 samples which is suitable for this simulation.

The uniform RNG provides random numbers in the range of [0,1]. The symbols that need to be generated for modulation need to be in the form of an integer in the range from 0 to 3. The random number to integer conversion is shown in Table 11.

Table 11: RND Number to Symbol Conversion

Random Number	Symbol
$0 < I_{j+1} < 0.25$	0
$0.25 < I_{j+1} < 0.5$	1
$0.5 < I_{j+1} < 0.75$	2
$0.75 < I_{j+1} < 1$	3

7.1.2 Modulator

A QPSK modulator can be represented by the constant envelope signal,

$$s_i(t) = \sqrt{\frac{2E_s}{T_{sym}}}\cos\left(2\pi f_c t + \frac{2\pi i}{4}\right) \tag{7.1}$$

where E_s is the symbol energy, T_{sym} is the symbol duration and i = 0, 1, 2, and 3 is the symbol to be transmitted. The symbols are generated using a uniform RNG. The carrier power is:

$$C = \frac{E_s}{T_{sym}} \tag{7.2}$$

For simulation purposes, it is convenient to normalize the carrier power to

$$C' = E_s \tag{7.3}$$

which is done by removing the T_{sym} in (7.1) resulting in:

$$s_i(t) = \sqrt{2E_s}\cos\left(2\pi f_c t + \frac{2\pi i}{4}\right) \tag{7.4}$$

The symbol energy is normalized to $E_s = 2$, which allows for simple calculations of E_b/N_o since $E_s = 2E_b$, making $E_b = 1$.

A fast fourier transform is performed on the sampled baseband data which returns the spectral information shown in Fig. 7.1. It is characterized by a main lobe with a bandwidth of 25MHz and some smaller sidelobes.

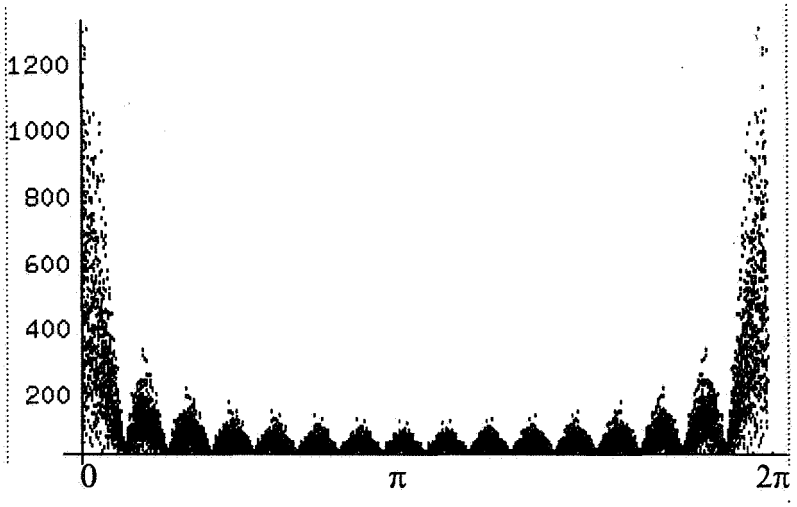


Fig. 7.1 Spectrum of the Sampled Baseband Signal

Computer simulation can not produce a true analog signal. The analog signal is represented with 16 samples per symbol. There is negligible increase in performance if a greater number of samples per symbol are used. A sample of the modulated signal is shown in Fig. 7.2.

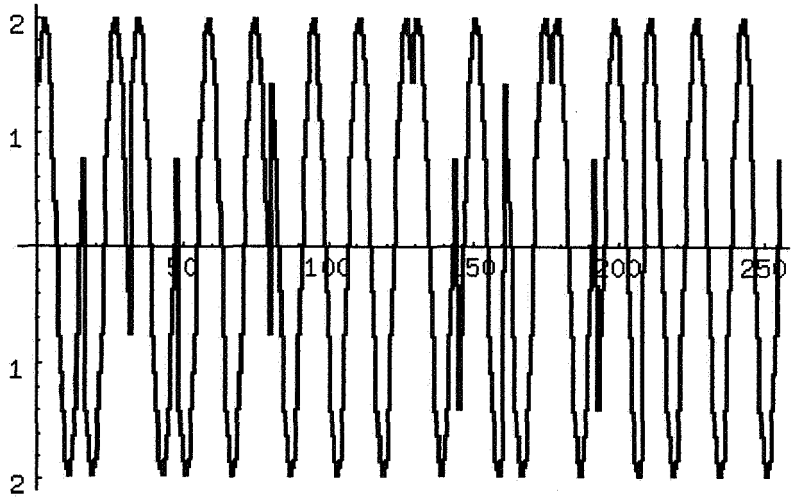


Fig. 7.2 QPSK Signal

The fast fourier transform is used to find the spectral information of the sampled analog signal. The spectrum of the pure modulated signal is shown

in Fig. 7.3. It is a shifted version of the baseband spectrum shown in Fig. 7.1. The main lobe is centered at f_c=25MHz and has a bandwidth of 50MHz.

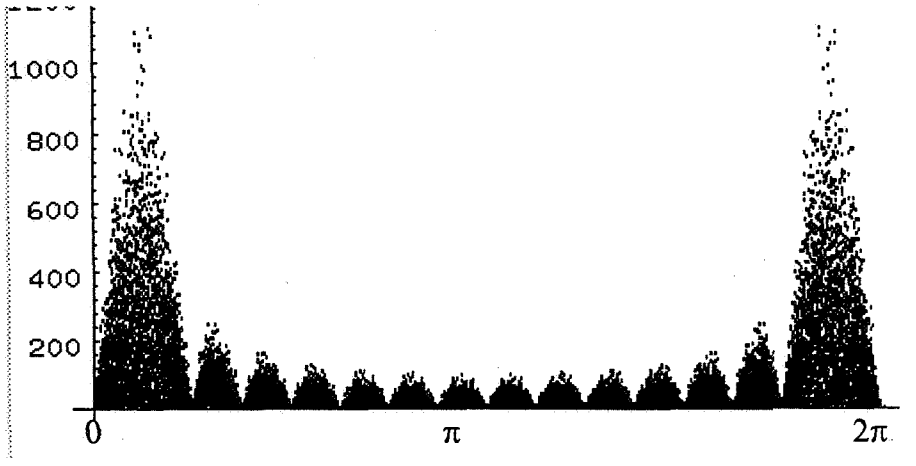


Fig. 7.3 Spectrum of the Modulated QPSK Signal

7.1.3 Bandpass Filter

Analog filters are required at the receive end of a satellite system. Bandpass filters are used to get rid of the unwanted noise that exists outside of the band of the desired signal. Bandlimiting is also used to alleviate interference caused by other signals in the channel.

An ideal bandpass filter is used in this simulation. It is designed to only pass the main lobe of the signal's spectrum. The filtered signal's spectrum is shown in Fig. 7.3. The Ideal bandpass filter is given by:

$$|H(f)| = 1 \quad \text{for } 0 \le f \le 50\text{MHz}$$
$$0 \quad \text{otherwise}$$

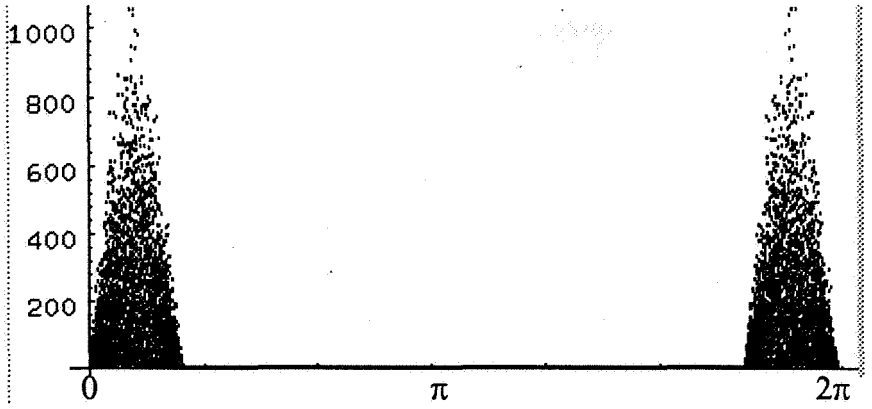


Fig. 7.4 Bandpass Filtered Spectrum

7.1.4 Sampler

After the bandpass filter, an inverse fast fourier transform is performed on the spectrum to return the signal back to the time domain. This signal with 16 samples per symbol is then sampled to produce a signal with 4 samples per symbol and is used as an input to the demodulator. This is done by using every fourth sample to describe the modulated signal. The spectrum of the signal with 4 samples per symbol is shown in Fig. 7.5. This signal is centered at f_C=25MHz and has a bandwidth of 50MHz. The sampling rate is 100MHz which satisfies the Nyquist theorem.

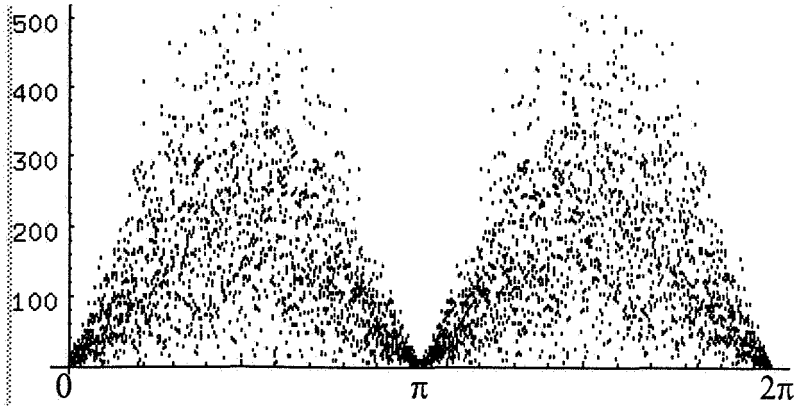


Fig. 7.5 Spectrum of the 4 Samples Per Symbol Signal

7.1.5 Additive White Gaussian Noise

To make the simulation as realistic as possible, the channel is chosen to be an additive white gaussian noise channel (AWGN). AWGN samples have an infinite variance (power) and a mean value of zero. Since it is impossible to simulate an infinite variance that is required to characterize the noise, another means of adding noise to the simulation is needed. When AWGN samples go through a filter, they become correlated and the noise variance will become finite. In reality, the modulated signal and the AWGN noise will be passed through a receive filter. For simulation purposes, the noise can be added after the receive filter with a finite variance as long as it is added in the frequency band that the filter passes.

The only noise of interest for simulation is the noise that has frequencies

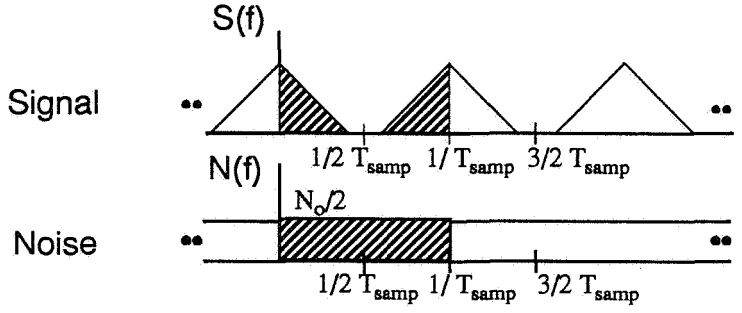


Fig. 7.6 Power In One Sample Period

which are less than $1/T_{samp}$ where T_{samp} is the sampling interval. The signal and noise power obtained in a sample period can be illustrated in Fig. 7.6. Let the signal power be denoted as S and the noise power be denoted as N. The signal power depends on the amount of energy per symbol, E_{sym}, added at the modulator end. The noise is AWGN and has a variance of $N_o/2$ where $N_o/2$ is the power spectral density of the noise. Over the simulation bandwidth, the amount of

noise power is:

$$N = N_o / 2\, T_{samp} \tag{7.5}$$

The carrier power is normalized by T_{sym} in (7.3). The C/N ratio must remain the same, so the noise variance is also scaled by the same factor to maintain this ratio. Therefore, the scaled noise variance used in the simulation is

$$N' = N_o T_{sym} / 2 T_{samp} \tag{7.6}$$

which can be written as

$$N' = N_o R / 2 \tag{7.7}$$

where $R = T_{sym}/T_{samp}$ is the number of samples per symbol.

Gaussian samples can be generated using the Box-Muller method [14] with a mean μ and a standard deviation σ with the following equation:

$$NORM(\mu, \sigma) = \sigma \sqrt{-2ln(RND)} \cos(RND) + \mu \tag{7.8}$$

where RND is a uniform random number generator in the range of [0,1], and $\sigma = N'^{1/2}$.

7.1.6 Demodulator

The demodulation is done by multiplying the modulated signal by

$$\sqrt{\frac{2}{T_{sym}}} \cos(2\pi f_c t) \tag{7.9}$$

and

$$\sqrt{\frac{2}{T_{sym}}} \sin(2\pi f_c t) \tag{7.10}$$

A small portion of the inphase channel samples after downconverting is shown in

Fig. 7.7.

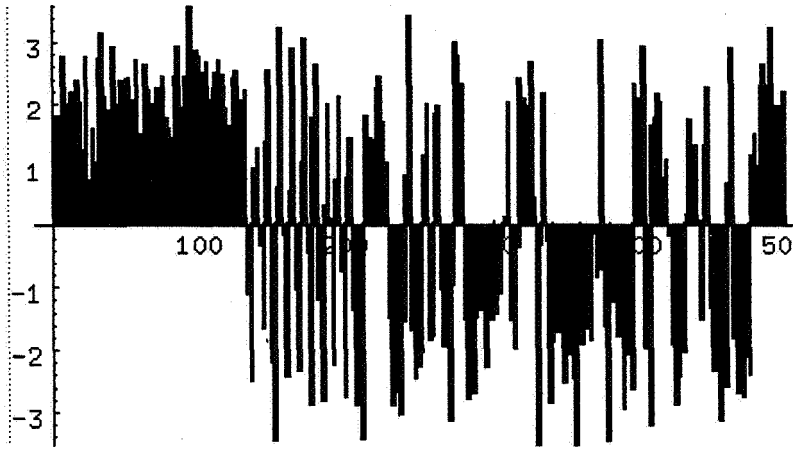


Fig. 7.7 Inphase Samples

The inphase and quadrature samples are then filtered and integrated over a single period. A decision is then made as to what bit has been transmitted. An example of the output of the demodulator is shown in Fig. 7.8 where the 32 PR symbols, 20 alternating TR symbols, the 15 UW symbols, and the data symbols are noticed.

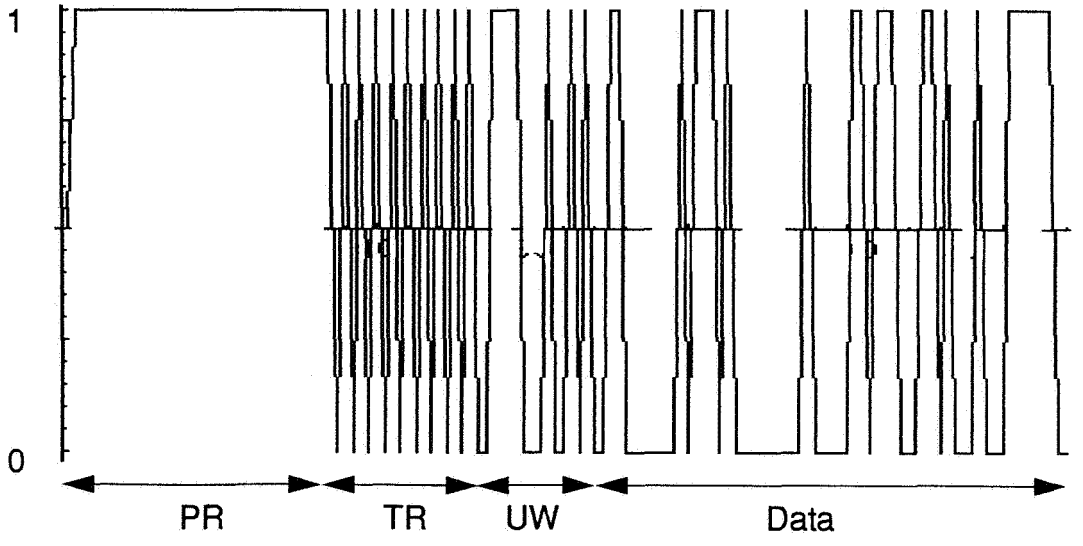


Fig. 7.8 Inphase Signal Decisions

7.2 VHDL Register Level Simulation

In order to more accurately represent the demodulator system, it had to be simulated in VHDL. This will allow a more precise representation of all of the components and how they will work together. A behavioral model written in the VHDL language describes the operation as well as the delay of a component. Many VHDL components can be connected together to form a circuit structure which can then be simulated.

7.2.1 VHDL Representation

VHDL code is written such that it will emulate the functionality of a layout component. If this simulates properly then there will be no doubt that the functionality of the system will be verified.

An example of the level of VHDL representation used in this is the adder cell. An adder module could have been written such that it could add together any word size, but will not represent the functionality of the layout component. A better way of representing it would be to create an adder cell with VHDL that would function the same as the layout adder cell. Simplified code for a VHDL adder cell is shown below:

```
Library unix;
Use unix.math.ALL;
Library lsim.terminals.ALL;
Use lsim.pragmas.ALL
ENTITY adder_cell IS
PORT(a, b, cin : IN LSIM_LOGIC; sout, cout : OUT LSIM_LOGIC);
END adder_cell;
ARCHITECTURE bhv OF adder_cell IS
FUNCTION sum (aa, bb, cc : LSIM_LOGIC) RETURN LSIM_LOGIC IS
```

```
BEGIN
RETURN (aa AND bb AND cc) OR (aa AND (NOT bb) AND (NOT cc)) OR (NOT cc) OR ((NOT aa)
AND (NOT bb) AND cc);
END sum
FUNCTION carry (aa, bb, cc : LSIM_LOGIC) RETURN LSIM_LOGIC IS
BEGIN
RETURN (aa AND bb) OR (aa AND cc) OR (bb AND cc);
END carry
BEGIN
sout <= sum (a, b, cin);
cout <= carry (a, b, cin);
END bhv
```

To create an n-bit adder, n of these adder modules will have to be connected in series. This can be done by writing a VHDL structure or by using the Mentor Graphics Led graphical interface. The graphical interface method is much simpler and less time consuming, so this is the route that is taken.

An icon must be created for the VHDL adder code. The inputs and outputs of the icon should be the same names as used in the code. An example of the icon used for the adder_cell is shown in Fig. 7.9.

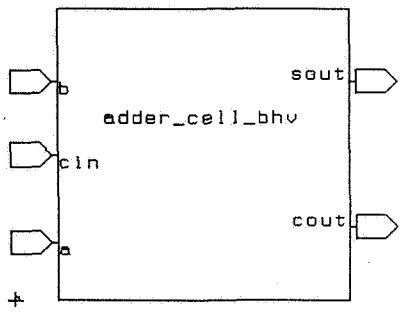


Fig. 7.9 Icon of Adder Cell

Notice that the input and output names correspond to the names given in the VHDL code. This icon can be called as an instance into another cell where it can

be wired to other cells. An 8-bit adder is shown in Fig. 7.10.

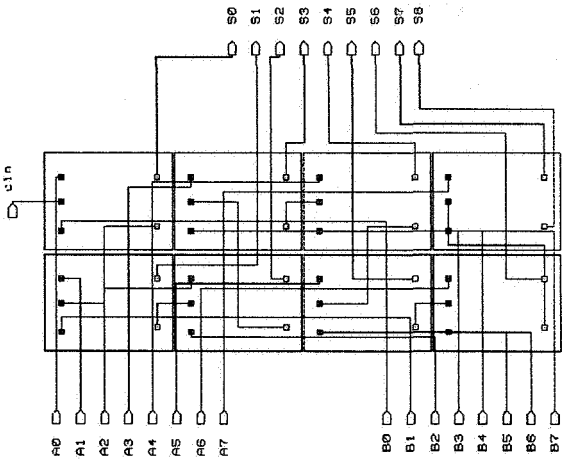


Fig. 7.10 Ripple Carry Adder Representation (8-bit)

The input and output terminals must be propagated before simulating. The names of these new terminals can be arbitrary. A netlist must be written from the graphical interface. This netlist will be simulated with VHDLsim, which is very similar to Lsim except that it will look for the compiled library in the parts directory. If the VHDL adder_cell code is saved into a file called adder.V, then the adder_cell icon must be named adder_bhv since this is what is in the compiled library in the parts directory.

The entire demodulator system is constructed using this method. This involved a lot of time, but the results are satisfying. An example of the VHDL model of the low pass filter is shown in Fig. 7.11.

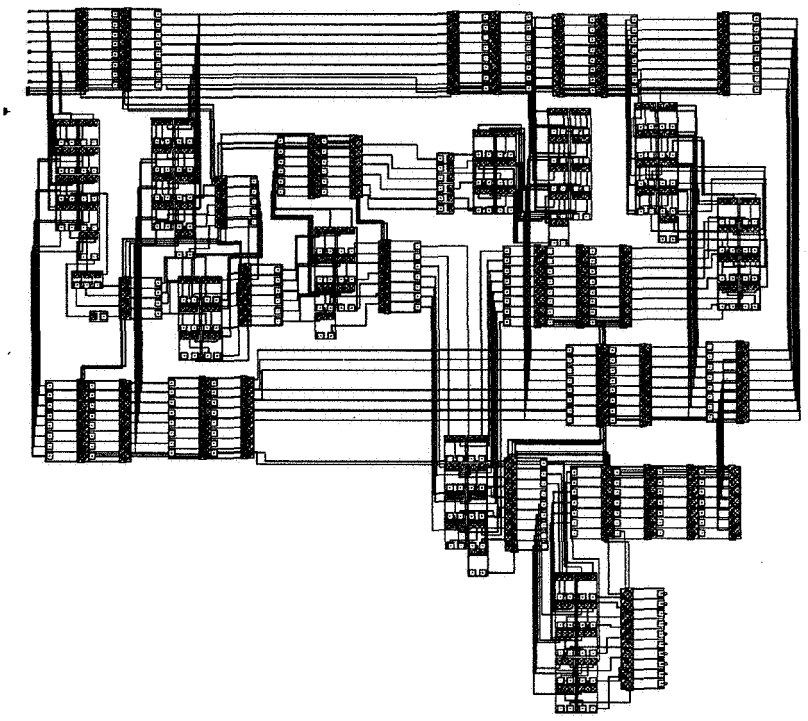


Fig. 7.11 VHDL Low Pass Filter

7.3 Layout Simulation

The layout simulation is done on each sub-component using Lsim. In this stage, proper sizing of the transistors is done to allow for a proper operation of the component under certain loads. The delay information is then extracted from the layout simulation and incorporated into the VHDL simulation as a look-up table. This technique allows the VHDL model to perform exactly as the physical layout will perform.

7.4 Symbol Error verses E_b/N_o Simulation

Various techniques are used to simulate the demodulator in this research. One way to check the performance of the chip is to find the probability of symbol error, P_e, given an E_b/N_0. The total number of symbols used in these

simulations are varied from 2^{10} symbols to 2^{17} symbols. In order to obtain a fair simulation, the number of symbols that need to be generated should be greater than $10/P_e$. This simply states that there should be at least 10 errors generated in the simulation interval of $10/P_e$ symbols. The results will be much better if the number of symbols is increased to greater than
$50/P_e$. P_e is calculated at various E_b/N_o and compared to a theoretical P_e curve for QPSK, which is given by:

$$P_e = \text{erfc}\left(\sqrt{\frac{E_b}{N_o}}\right) \tag{7.11}$$

where

$$\text{erfc}(u) = \frac{2}{\sqrt{\pi}}\int_u^{\infty} e^{-z^2} dz \tag{7.12}$$

The actual curves generated for the QPSK and the SQPSK are the same, so only one curve is illustrated. As shown in the signal of Fig. 7.4, only the main lobe is passed by the bandpass filter. The filtered spectrum is then inverse fourier transformed and sampled at 4, 8, and 16 samples per symbol and simulated for different E_b/N_o. The results are shown in Fig. 7.12.

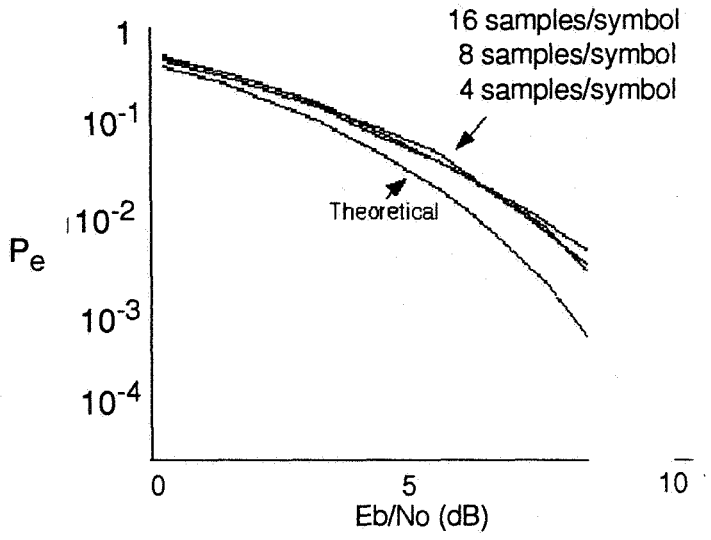


Fig. 7.12 P_e Simulation Results

These results show that the performance is the same no matter how many samples per symbol are used to represent the signal. This is because increasing the number of samples per symbol gives no more information about the bandlimited signal.

The system degradation is caused by the bandlimiting of the signal with the bandpass filter. To illustrate this, the bandwidth of the ideal bandpass filter is increased from passing only the main lobe to passing the main lobe plus a number of sidelobes. This simulation is performed with a signal represented by 4 samples per symbol. The results are shown in Fig. 7.13.

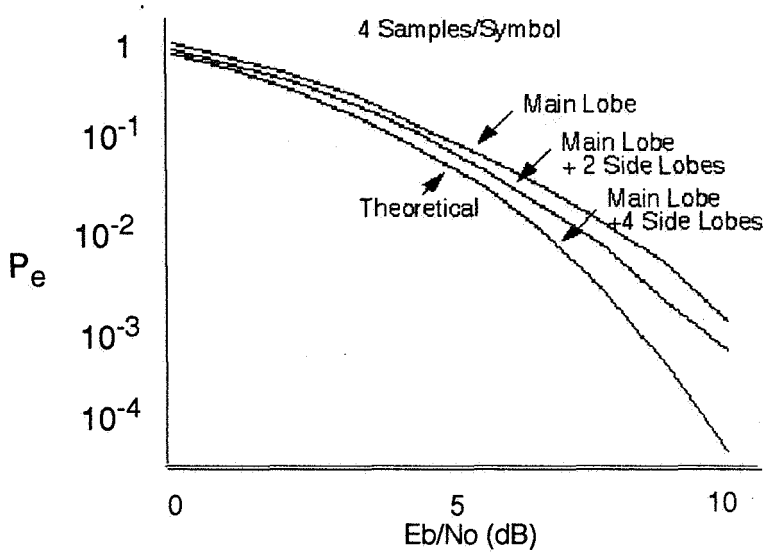


Fig. 7.13 Affects of Increasing The Bandwidth of the Bandpass Filter

These results demonstrate that with increased signal bandwidth, the performance will be improved. The reason for this is that the correlation receiver is perfectly matched to a pure QPSK modulated signal. Therefore, it acts as an optimal filter. As soon as the signal is bandlimited, the correlation receiver is no longer matched to the signal and becomes sub-optimum. As the bandwidth of the bandpass filter is increased, the response of the combination of filters approaches the response of the matched filter which improves performance.

In practice, a communication system is designed to use the smallest possible bandwidth. An extra 2dB of power must be provided to achieve a 10^{-6} P_e, compared to the theoretical power level needed for this error rate, in a carefully filtered QPSK link [10].

Chapter 8 Conclusions

8.0 Conclusions

A single chip QPSK/SQPSK demodulator has been developed using the 0.8μ CMOS technology. All design specifications are implemented successfully. Modifications have been made to a previous architecture to enhance the performance and to decrease the area of the demodulator. The final area of the demodulator is 2.0 x 2.6 mm^2, which will have no problems fitting in the chip area assumed to be 1.4 x 1.4 cm^2.

Simulation of the demodulator has been done in using a functional level representation, a behavioral level representation, and a component level layout simulation. Verification of the block placement and wiring has also been accomplished using a test vector approach.

A key part of the research is the development of the component generators. Generators are developed for the adder, multiplier, ROM and SAM. These design automation techniques are used often and decreased the design time considerably.

The VLSI design of the demodulator is feasible. Degradation caused by bandlimiting the signal will require that the signal power be increased by about 2dB in order to operate with a practical probablilty of symbol error of $P_e=10^{-6}$. This is usually the case anyhow for a carefully filtered QPSK channel.

8.1 Future Research

Time is an important factor when designing large VLSI systems such as a demodulator. Knowledge may be gained in the middle of the research that would improve the system greatly, but there is no time to change the existing architecture. Some of the findings that may improve the system are covered in the following sections.

8.2 Timing Recovery

A more reliable timing recovery unit should be used in further research. Although a little amount of filtering is done in the channel detection section of the TRU, there are better algorithms that will enhance the system performance and deal with noise a little better. Two papers of interest are [15] and [16]. The first presents a TRU that uses only one sample per symbol to extract the timing information. This TRU must be present in a directed decision system. The second presents a TRU that does it's decision with two samples per symbol. It can be used both at baseband or at bandpass levels.

8.3 8PSK and 16QAM

NASA has encouraged further research into a couple of different modulation schemes. These are the 8-PSK and the 16-QAM modulation schemes.

8-PSK is another phase shift keyed modulation scheme. It has a greater bandwidth efficiency than QPSK at the expense of more transmitted power needed. The architecture is not much different from that of a QPSK demodulator.

The envelope of the signal is constant so the constellation of the 8-PSK signal is still circular as in the QPSK case. A block diagram of an 8-PSK demodulator is shown in Fig. 8.0.

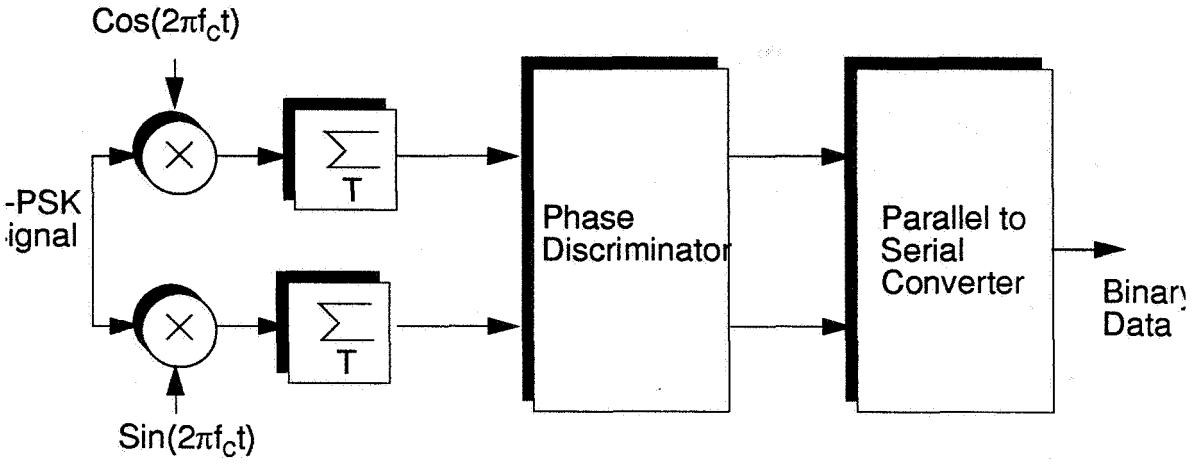


Fig. 8.0 8PSK Demodulator

16-QAM is a combination of phase shift keying and amplitude modulation. It enables the transmission of $M=L^2$ independent symbols. Since the amplitude is not constant, the constellation is no longer circular. The constellation now becomes square. Fig. 8.1. shows a block diagram of an M-ary QAM demodulator.

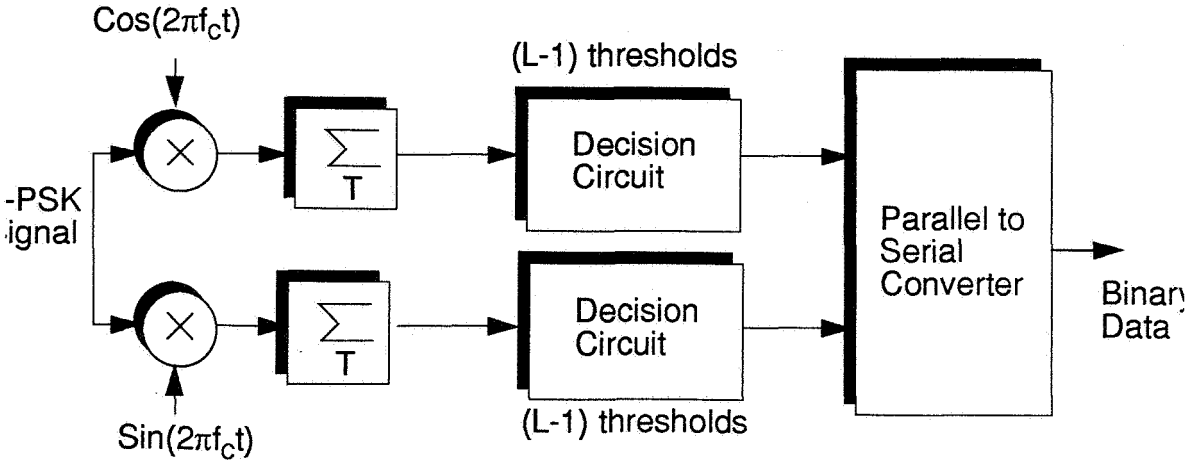


Fig. 8.1 16QAM Demodulator

For more information on these two modulation schemes, see [3].

8.4 Assume Off-Chip Analog Downconversion

In many applications, the IF rate will be above 70MHz. Today, the CMOS technology is unable to handle such high sampling rates. So an off chip analog downconversion unit should be assumed in further research. This will simplify the system and the simulation of the system.

8.5 Baud Rate = IF

If the baude rate (symbol rate) is equal to the IF signal, then there is no need for a down conversion unit. Consider the sinusoids shown in Fig. 8.2.

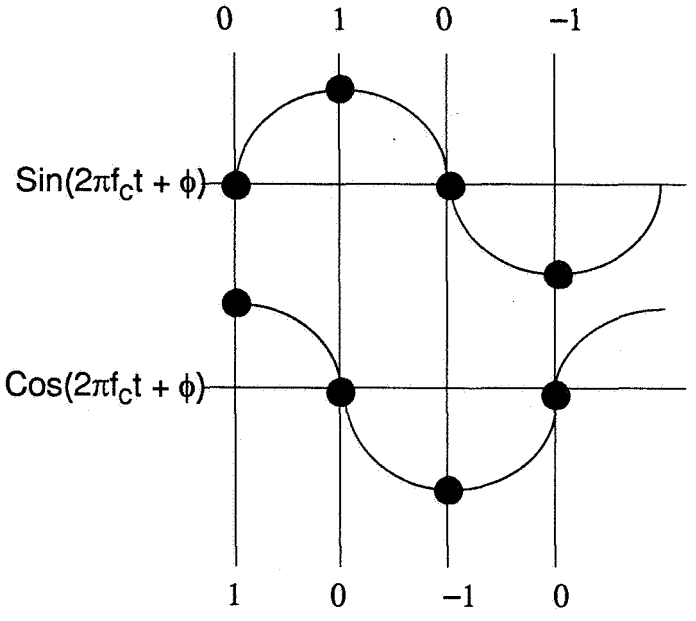


Fig. 8.2 Sinusoidal Samples

If the symbol is sampled 4 times then the oscillator samples become $\cos(n\pi/2)$ and $\sin(n\pi/2)$. The values of these sinusoids are 0, 1, 0, -1, which are shown in Fig. 8.2. Therefore, there is no need for a multiplier. All that is needed is a

demultiplexer and an inversion control unit [17] as shown in Fig. 8.3.

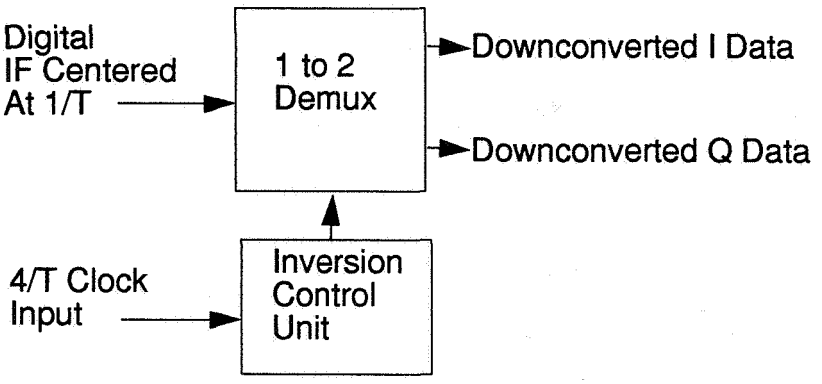


Fig. 8.3 Simple Downconversion Unit

8.6 Simulation with BOSS, SPW,...

It would be a good idea to use BOSS or SPW to generate test vectors to be used as inputs to the system. These packages have ready-made modules that can be used to generate the proper signals with noise. Then the performance can be compared to the SPW simulation.

Appendix A Generator Code

A.0 Contents

A template for writing code for generators is offered in section A.1. This template shows an outline of how the generators were developed for this project. The sections following this template present the code for each individual generator.

The generator code was written in Lx. Lx is a procedural interface to the L database and Led graphics editor. It is built from a set of database interface functions and from general purpose language called GENIE. Lx provides access to the information within the L database and provides interaction with the Led graphis editor.

A.1 Code Template for a Generator

This is an example of a template that can be used in developing generators. In order to get a better grasp as to what is going on, the Lx manual should be consulted before reading this code.

func gen ((int a b)(string c d)(float e f)) { Define the function.

*int g h*Variable assignment.

float i j

list save Define the list that the cells will be appended to.

abstract Lid A B C Define the Lids.

lread cellA Read in the cells

lread cellB that will be used in

lread cellC the generator.

A = (set_cell cellA) Assign the Lids to

B = (set_cell cellB) a cell.

C = (set_cell cellC)

The algorithm for placement of the cells should be placed here. The algorithm can be implemented using logical statements such as FOR, IF and CASE. To create an array of cells, the append statement is used.

append @save A

This appends the cell A to a list called save. The next cell that is appended to the list will be connected to the cell A. Once the algorithm part is finished, end the function with a "}"

}

After this is written, the file should be saved as gen.m. To run this program, go to the command prompt in Led and type:

load gen.m

gen 1 2 hello goodbye 1.0 2.0

The parameters after gen need to match the types that are specified in the func statement. The generator will then read in the cells and place them according to some algorithm.

A.2 Adder Generator Code

This is the Lx code for the layout of the adder. The adder is a ripple carry adder that is designed using the transmission gate adder cell. User specifications are: 1.) size = n, for an n bit adder. 2.) name of the adder cell.

Beginning of Program

```
func adder ((int size)(string name)) {
```

Declarations

```
string b a c
int i count k check div
list lst
abstract Lid up down ad source sinc upflip downflip t source2 sinc sinc2
if((@size % 2) == 0) {
div = (@size / 2)
} else {
div = ((@size + 1) / 2)
}
```

Initialization of variables and assignment of abstract Lids to layout cells

```
check = 0
count = 0
lread adup.L
lread adupflip.L
lread addown.L
lread addownflip.L
lst = '()
up = (set_cell adup)
upflip = (set_cell adupflip)
down = (set_cell addown)
downflip = (set_cell addownflip)
```

Start a new layout cell called name

```
set_cell_id (add_cell @LAYOUT @name)
```

Append the top set of cells

```
for(i = 0; @i < @div; i++) {
                if((@i % 2) == 0) {
                append lst @up
                } else {
                append lst @upflip
                }
}
```

Append the bottom set of cells

```
for(i = 0; @i < @div; i++) {
                if((@i % 2) == 0) {
                append lst @down
                } else {
                append lst @downflip
                }
}
```

Create the layout

```
ad = (add_array  @lst 2 @div)
```

Add wires and top level connectors

Wire bottom carry lines

```
if((@div % 2) == 0) {
for(i = 0; @i < @div; i += 2) {
source = (make_target out0 @ad 1 @i)
sinc = (make_target in0 @ad 1 (@i + 1))
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 13.35)(make_seg @VER 0.0)(make_seg @HOR 0.0))
}
} else {
for(i = 0; @i < (@div - 1); i += 2) {
source = (make_target out0 @ad 1 @i)
sinc = (make_target in0 @ad 1 (@i + 1))
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 13.35)(make_seg @VER 0.0)(make_seg @HOR 0.0))
}
}
if(@size >= 6) {
```

Wire top carry lines

```
if(@size == 6) { k = 1 }
if(@size == 8) { k = 1 }
if(@size == 10) { k = 2 }
if(@size == 12) { k = 2 }
if(@size == 14) { k = 3 }
if(@size == 16) { k = 3 }
for(i = 0; @i < (@k * 2); i += 2) {
source = (make_target out0 @ad 0 (@i + 1))
sinc = (make_target in0 @ad 0 (@i + 2))
```

```
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@VER 2.4)(make_seg @HOR 57.4)(make_seg @VER -2.4))
}
}
```

Add top level connectors to the top set of inputs

```
for(i = 0; @i < @div; i++) {
k = 0
if((@i % 2) == 0) {
                count = (@i * 2)
} else {
                count = ((@i * 2) + 1)
}
b = (cat "b[" @count "]")
a = (cat "a[" @count "]")
count += 1
source = (make_target B @ad @k @i)
source2 = (make_target A @ad @k @i)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @b)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source2) @R0 0 @source2 @a)
}
```

Add top level connectors to the bottom set of inputs

```
for(i = 0; @i < @div; i++) {
k = 1
if((@i % 2) == 0) {
count = ((@i * 2) + 1)
} else {
count = ((@i * 2))
}
b = (cat "b[" @count "]")
a = (cat "a[" @count "]")
count += 1
source = (make_target B @ad @k @i)
source2 = (make_target A @ad @k @i)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @b)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source2) @R0 0 @source2 @a)
}
```

Add top level connectors to the outputs

```
count = 0
for(i = 0; @i < @div; i++) {
b = (cat "s[" @count "]")
count += 1
a = (cat "s[" @count "]")
count += 1
source = (make_target S0 @ad 0 @i)
source2 = (make_target S1 @ad 0 @i)
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source) @R0 0 @source @b)
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source2) @R0 0 @source2 @a)
}
```

Add top level connector for Cin input

```
b = "cin[0]"
source = (make_target in0 @ad 0 0)
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source) @R0 0 @source @b)

if((@div % 2) == 0) {
b = (cat "s[" @size "]")
source = (make_target out0 @ad 0 (@div - 1))
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source) @R0 0 @source @b)
} else {
b = (cat "s[" @size "]")
source = (make_target out0 @ad 1 (@div - 1))
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source) @R0 0 @source @b)
}

if((@div % 2) == 0) {
source = (make_target out2 @ad 0 (@div - 1))
sinc = (make_target out5 @ad 1 (@div - 1))
add_wire @source @sinc (get_named_type MET3) (bld (make_seg \
@HOR 3.4)(make_seg @VER -23.6)(make_seg @HOR -3.4))
t = (add_terminal (get_named_type VDD) (get_named_type MET3) (get_loc
@source) @R0 0 @source)
```

Wire the Vdd and GND lines and add the top level terminals

```
source = (make_target in3 @ad 0 0)
sinc = (make_target in5 @ad 0 0)
```

```
add_wire @source @sinc (get_named_type MET3) (bld (make_seg \
@HOR -3.4)(make_seg @VER -20.0)(make_seg @HOR 3.4))
t = (add_terminal (get_named_type GND) (get_named_type MET3) (get_loc
@sinc) @R0 0 @sinc)
source = (make_target in5 @ad 1 0)
add_wire @source @sinc (get_named_type MET3) (bld (make_seg \
@HOR -3.4)(make_seg @VER 23.6)(make_seg @HOR 3.4))
} else {
source = (make_target out1 @ad 0 (@div - 1))
sinc = (make_target out3 @ad 1 (@div - 1))
add_wire @source @sinc (get_named_type MET3) (bld (make_seg \
@HOR 3.4)(make_seg @VER -23.6)(make_seg @HOR -3.4))
t = (add_terminal (get_named_type VDD) (get_named_type MET3) (get_loc
@source) @R0 0 @source)
source = (make_target in3 @ad 0 0)
sinc = (make_target in5 @ad 0 0)
add_wire @source @sinc (get_named_type MET3) (bld (make_seg \
@HOR -3.4)(make_seg @VER -20.0)(make_seg @HOR 3.4))
t = (add_terminal (get_named_type GND) (get_named_type MET3) (get_loc
@sinc) @R0 0 @sinc)
source = (make_target in5 @ad 1 0)
add_wire @source @sinc (get_named_type MET3) (bld (make_seg \
@HOR -3.4)(make_seg @VER 23.6)(make_seg @HOR 3.4))
}
}
```

A.3 Multiplier Generator Code

This is the Lx code for the layout of the multiplier. The multiplier is constructed following the Bough/Wooley algorithm. It is a 2's compliment multiplier. [illegible] be modified to generate m by n multipliers. User specifications are:

1.) size = n, for an n by n multiplier.

2.) name of the multiplier cell.

Beginning of Program

```
func mult ((int size)(string name)) {
```

Declarations

```
list lst
int i j count tmp piper tmp1
string a b
abstract Lid space a1 a2 a3 a4 m1 m2 m3 m4 m5 x mult
abstract Lid source sinc source2 sinc2 t p ps1 pst
```

Read in the cells

```
lread pipe.L
lread pipeside1.L
lread pipesidetop.L
lread mult_cell.L
lread mult2_cell.L
lread mult3_cell.L
lread mult4_cell.L
lread mult5_cell.L
lread and_cell.L
lread and2_cell.L
lread and3_cell.L
lread and4_cell.L
lread mult_space.L
lread xorplusor.L
```

Initialize the variables and assign abstract Lids to layout cells

```
lst = '()
piper = 5
p = (set_cell pipe)
ps1 = (set_cell pipeside)
pst = (set_cell pipetop)
m1 = (set_cell mult)
m2 = (set_cell mult2)
m3 = (set_cell mult3)
m4 = (set_cell mult4)
m5 = (set_cell mult5)
a1 = (set_cell and)
a2 = (set_cell and2)
a3 = (set_cell and3)
a4 = (set_cell and4)
x = (set_cell xorplusor_L)
space = (set_cell mult_space)
```

Start a new layout cell and call it name

```
set_cell_id (add_cell @LAYOUT "@name")
```

Begin placing the cells

```
append lst @space
for(i = 0; @i < (@size - 1); i++) {
        append lst @a4
}
for(i = 0; @i < (@size - 1); i++) {
        if(@i == @piper) { append lst @pst
        } else {
                if(@i > @piper) {
                        append lst @ps1
}               else {
                        append lst @a1 }
}
for(j = 0; @j < (@size - 1); j++) {
        if(@i == @piper) {
                append lst @p
        } else {
                append lst @m1 }
        }
}
append lst @x
for(i = 0; @i < (@size - 1); i++) {
        append lst @m2
}
append lst @m4
for(i = 0; @i < (@size - 2); i++) {
append lst @m3
}
append lst @m5
```

Create a layout of the multiplier

```
mult = (add_array  @lst (@size + 2) @size)
```

Wire the Vdd and GND terminals and place top level connectors for Vdd, GND, Inputs and Outputs.

Wire vdd and gnds

```
source = (make_target vdd1 @mult 0 (@size - 1))
source2 = (make_target gnd0 @mult 0 (@size - 1))
t = (add_terminal (get_named_type VDD) (get_named_type MET3) (get_loc
@source) @R0 0 @source)
t = (add_terminal (get_named_type GND) (get_named_type MET3) (get_loc
```

```
@source2)
@R0 0 @source2)

count = 1
for(i = 0; @i < (@size - 1); i ++) {
source = (make_target vdd1 @mult @count (@size - 1))
source2 = (make_target gnd1 @mult @count (@size - 1))
t = (add_terminal (get_named_type VDD) (get_named_type MET3) (get_loc
@source) @R0 0 @source)
t = (add_terminal (get_named_type GND) (get_named_type MET3) (get_loc
@source2)
@R0 0 @source2)
count += 1
}
```

Add top level connectors for Vdd and GND

```
source = (make_target vdd1 @mult (@size) (@size - 1))
source2 = (make_target gnd0 @mult (@size) (@size - 1))
t = (add_terminal (get_named_type VDD) (get_named_type MET3) (get_loc
@source) @R0 0 @source)
t = (add_terminal (get_named_type GND) (get_named_type MET3) (get_loc
@source2) @R0 0 @source2)
```

Add terminals to the clock inputs

```
a = "clkin"
b = "clkout"
source = (make_target clkin @mult (@piper + 1) 0)
source2 = (make_target clkout @mult (@piper + 1) (@size - 1))
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source) @R0 0 @source @a)
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source2) @R0 0 @source2 @b)

count = 0
```

Add terminals to the inputs

```
tmp = (@size - 1)
a = (cat "x[" @tmp "]")
b = (cat "y[" @count "]")
source = (make_target in3 @mult 0 0)
source2 = (make_target in4 @mult 0 0)
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source) @R0 0 @source @a)
```

```
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source2) @R0 0 @source2 @b)

count = 1
for(i = 0; @i < (@size - 1); i ++) {
tmp = (@size - 1 - @count)
a = (cat "x[" @tmp "]")
source = (make_target in0 @mult 0 @count)
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source) @R0 0 @source @a)
count += 1
}
count = 1
for(i = 0; @i < (@size - 2); i ++) {
b = (cat "y[" @count "]")
source = (make_target in4 @mult @count 0)
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source) @R0 0 @source @b)
if(@i == (@piper - 1)) { count += 1}
count += 1
}
count = (@size)
b = (cat "y[" @count "]")
source2 = (make_target b @mult (@size) 0)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source2) @R0 0 @source2 @b)
```

Add terminals to the outputs

```
count = 1
for(i = ((2 * @size) - 3); @i > (@size - 2); i--) {
tmp1 = @i
if(@i == (@size - 1)) { tmp1 = (@size - 1)}
b = (cat "p[" @tmp1 "]")
source = (make_target S0 @mult (@size + 1) @count)
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source) @R0 0 @source @b)
count += 1
}
tmp = ((2 * @size) - 1)
b = (cat "p[" @tmp "]")
tmp = ((2 * @size) - 2)
a = (cat "p[" @tmp "]")
source2 = (make_target out7 @mult (@size + 1) 0)
source = (make_target S0 @mult (@size + 1) 0)
t = (add_terminal (get_named_type OUT) (get_named_type MET1) (get_loc
```

```
@source2) @R0 0 @source2 @b)
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source) @R0 0 @source @a)

}
```

A.4 Read ROM Array Cells

This code is loaded in by the ROM generator. It will allow this generator to read in the cells needed for the ROM array.

```
func read_rom {
lread rom_1st_beginspace2_L
lread rom_nload_down_tranc_L
lread rom_nload_down_tran_L
lread rom_end_tran_L
lread rom_p_up_L
lread rom_p_up_space_L
lread rom_p_down_L
lread rom_p_down_space_L
lread rom_fbs_L
lread rom_lbs_L
lread rom_bs_L
lread rom_1st_gnd_down_L
lread rom_last_gnd_down_L
lread rom1stend_L
lread rom_1st_beginspace_L
lread romlastend_L
lread rom_last_beginspace_L
lread rom_last_beginspace2_L
lread rom_1st_gnd_up_L
lread rom_1st_gnd_middle_L
lread rom_last_gnd_up_L
lread rom_last_gnd_middle_L
lread rom_begin_space_L
lread rom_begin_space2_L
lread romend_L
lread rom_gnd_up_L
lread rom_gnd_middle_L
lread rom_gnd_down_L
lread up_notran1_L
lread up_tran1_L
lread down_notran_nocontact1_L
lread down_notran_contact1_L
lread down_tran1_L
```

```
}
```

A.5 Read Row Decoder Cells

This code is loaded in by the ROM generator. It will allow this generator to read in the cells needed for the row decoder.

```
func read_dec {
lread dec_gnd_begin_L
lread dec_gnd_middle_L
lread dec_gnd_end_L
lread dec_tran_longalone1_L
lread dec_tran_longalone2_L
lread dec_tran_longpoly_L
lread dec_tran_shtpoly_L
lread dec_tran_longpoly_middle1_L
lread dec_tran_longpoly_middle2_L
lread dec_tran_alone_L
lread dec_2_L
lread dec_22_L
lread dec_notran_L
}
```

A.6 ROM Array Generator Code

Beginning of Program

```
func rom (int l w word_size num_blocks) {
```

Declarations

```
list lst lst2
list sav lst3 re
int i j a b k count counter strap check check2 h count2
int top place times read_data mult tmp1 tmp2 cnt div
float inc x y
float pi time1 time2
string aa bb
file fp
```

```
abstract Lid z td_nt_nc q
abstract Lid td_nt_c td_t
abstract Lid tu_nt tu_t
abstract Lid bf bf_out
abstract Lid gnddown gndup gndmiddle end
abstract Lid beginspace beginspace2 firstgndup firstgndmiddle
abstract Lid lastgndup lastgndmiddle
abstract Lid firstbeginspace firstbeginspace2 first
abstract Lid lastbeginspace lastbeginspace2 last lastgnddown
abstract Lid firstgnddown
abstract Lid pass_up pass_down after_pu spacer_up
abstract Lid empty_up empty_down spacer_down fes les es
abstract Lid pass_up_nc pass_down_nc spacer_up_c after_pu_c
abstract Lid p_up_space p_up p_down p_down_space
abstract Lid fbs lbs bs pass fbs2 bs2 lbs2
abstract Lid source source2 sinc sinc2 t
abstract Lid loadup loaddown loaddownc loadupsp loaddownsp
```

Assign cells to abstract Lids

```
loaddown = (set_cell rom_nload_down_tran_L)
loaddownc = (set_cell rom_nload_down_tranc_L)
pass = (set_cell rom_end_tran_L)
fbs = (set_cell rom_fbs_L)
bs = (set_cell rom_bs_L)
lbs = (set_cell rom_lbs_L)
p_up = (set_cell rom_p_up_L)
p_down = (set_cell rom_p_down_L)
p_up_space = (set_cell rom_p_up_space_L)
p_down_space = (set_cell rom_p_down_space_L)
firstgnddown = (set_cell rom_1st_gnd_down_L)
lastgnddown = (set_cell rom_last_gnd_down_L)
firstbeginspace = (set_cell rom_1st_beginspace_L)
firstbeginspace2 = (set_cell rom_1st_beginspace2_L)
first = (set_cell rom1stend_L)
lastbeginspace = (set_cell rom_last_beginspace_L)
lastbeginspace2 = (set_cell rom_last_beginspace2_L)
last = (set_cell romlastend_L)
firstgndup = (set_cell rom_1st_gnd_up_L)
firstgndmiddle = (set_cell rom_1st_gnd_middle_L)
lastgndup = (set_cell rom_last_gnd_up_L)
lastgndmiddle = (set_cell rom_last_gnd_middle_L)
beginspace = (set_cell rom_begin_space_L)
beginspace2 = (set_cell rom_begin_space2_L)
end = (set_cell romend_L)
gndup = (set_cell rom_gnd_up_L)
```

```
gndmiddle = (set_cell rom_gnd_middle_L)
gnddown = (set_cell rom_gnd_down_L)
td_nt_c = (set_cell down_notran_contact1_L)
td_t = (set_cell down_tran1_L)
td_nt_nc = (set_cell down_notran_nocontact1_L)
tu_nt = (set_cell up_notran1_L)
tu_t = (set_cell up_tran1_L)
```

Define a new Layout cell

```
set_cell_id (add_cell @LAYOUT rom1)
```

Initialize variables

```
check = 0
div = 1
check2 = 0
strap = 10
count = 0
counter = 0
mult = 0
tmp1 = 0
tmp2 = 0
lst = ' ()
lst3 = ' ()
lst2 = ' ()
sav = '()
re = '()
x = 0.0
y = 0.0
a = 1
b = 0
top = 0
place = @w
times = 1
for(i = 0; @i < @w; i++) {
append sav @a
}

fp = (fopen nco_rom.dat r)
for(i = 0; @i < (@w * @l); i++) {
fscanf @fp "%d" read_data
append re @read_data
if((@i % 100) == 0) {
println @i
}
```

```
}
```

Place the load transistors

```
for(h = 0; @h < 1; h++) {
append lst @firstbeginspace2
for(k = 1; @k <= @w; k++) {
if((@k % 2) == 0) {
append lst @loaddownc
} else {
append lst @loaddown }
if(@k < @w) {
if((@k % @strap) == 0) {
append lst @beginspace2 }
}
}
append lst @lastbeginspace2
}
```

Place the precharge transistors.

```
for(h = 0; @h < 2; h++) {
if((@h % 2) == 0) {
append lst @fbs
for(k = 1; @k <= @w; k++) {
if((@k % 2) == 0) {
append lst @p_up_space
} else {
append lst @p_up }
if(@k < @w) {
if((@k % @strap) == 0) {
append lst @bs }
}
}
append lst @lbs
} else {
append lst @firstbeginspace
for(k = 1; @k <= @w; k++) {
if((@k % 2) == 0) {
append lst @p_down
} else {
append lst @p_down_space }
if(@k < @w) {
if((@k % @strap) == 0) {
append lst @beginspace }
}
```

```
}
append lst @lastbeginspace

}
}
```

Start placing the mem trans

```
for(k = 0; @k < 1; k++) {
for(i = 0; @i < @l; i++) {
println @i
if(@k == 0) {
if(@top == 0) {
if(@i == 0) {
                append lst @firstgndup
} else { append lst @firstgndmiddle }
} else { append lst @firstgnddown }
}

times = (@w * @i)
count = 0
for(j = 0; @j < @w; j++) {
count += 1
if(@re[(@j + @times)] == 0) {
if(@top == 0) { append lst @td_t
} else { append lst @tu_t }
place += 1
append sav @b
} else {
if(@top == 0) {
if((@sav[(@place - @w)]) == 0) {
                append lst @td_nt_c
} else { append lst @td_nt_nc }
} else { append lst @tu_nt }
place += 1
append sav @a
}
if(@count < @w) {
if((@count % @strap) == 0) {
if(@check == 0) {
counter++
}
if(@top == 0) {
if(@i == 0) {
                append lst @gndup
} else {append lst @gndmiddle }
```

```
} else { append lst @gnddown }
}
}
}
check = 1
count = 0
if(@k < 0) {
if(@top == 0) {
if(@i == 0) {
append lst @gndup
} else {append lst @gndmiddle }
} else { append lst @gnddown }
}
if(@k == 0) {
if(@top == 0) {
if(@i == 0) {
append lst @lastgndup
} else {append lst @lastgndmiddle }
} else { append lst @lastgnddown }
}
if(@top == 1) {
top = 0
} else { top = 1}
}
}
count = 0
mult = 0
append lst @first
for(k = 1; @k <= @w; k++) {
append lst @pass
count += 1
if((@count % @strap) == 0) {
count = 0
if(@mult < @counter) {
append lst @end }
mult ++
}
}
append lst @last
println @count
println @mult
println @counter
```

Create the layout of the ROM array

```
z = (add_array @lst (@l + 4) (@w + @counter + 2) (bld @x @y) @R0)
```

Terminals for VDD and GND

```
count = 0
source = (make_target in2 @z 0 0)
source2 = (make_target in4 @z 2 0)
t = (add_terminal (get_named_type VDD) (get_named_type MET2) (get_loc
@source) @R0 0 @source)
t = (add_terminal (get_named_type GND) (get_named_type MET2) (get_loc
@source2) @R0 0 @source2)
source = (make_target in2 @z 2 0)
t = (add_terminal (get_named_type VDD) (get_named_type MET2) (get_loc
@source) @R0 0 @source)

source = (make_target in6 @z 2 0)
aa = "rom_clk"
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @aa)
```

Terninals for inputs

```
count = 1
for(i = 0; @i < (@l / 2); i++) {
aa = (cat "rom_in[" @count "]")
count += 1
source = (make_target in2 @z (@count + 1) 0)
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source) @R0 0 @source @aa)
bb = (cat "rom_in[" @count "]")
count += 1
source2 = (make_target in1 @z (@count + 1) 0)
t = (add_terminal (get_named_type IN) (get_named_type MET2) (get_loc
@source2) @R0 0 @source2 @bb)
}

count2 = 1
count = 1
cnt = 1
for(j = 0; @j < @num_blocks; j++) {
for(i = 0; @i < (@word_size); i++) {
aa = (cat "rom_out[" @count2 "]")
source = (make_target out1 @z (@l + 3) @count)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @aa)
if((@cnt % @strap) == 0) {
count += 1
```

```
}
cnt += 1
count += 1
count2 += 1
}
}
count2 = 1
source = (make_target in3 @z (@l + 3) 0)
aa = (cat "coldec_in[" @count2 "]")
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @aa)

count = 1
for(i = 1; @i < (@num_blocks * @div); i++) {
count2 = (@count + 1)
aa = (cat "coldec_in[" @count2 "]")
source = (make_target in2 @z (@l + 3) ((@strap + 1) * @count))
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @aa)
count += 1
}


fclose @fp
}
```

A.7 Row Decoder Generator Code

Beginning of program

```
func dc ((int in rows flip)(float xx yy)) {
```

Declarations

```
int top col row integ k incr tran save_inc
int tmp count count2 a b i x
string aa bb
list lst sav compare
abstract Lid zz long short alone dc2 leftbuf rightbuf
abstract Lid dc1 notran longalone begin middle end
abstract Lid longalone1 longalone2 long_middle1 long_middle2
```

```
abstract Lid ckt1 ckt2 ckt3 buf_end buf_begin buf_left buf_right
abstract Lid buf_last sht
abstract Lid source source2 sinc sinc2 t
begin = (set_cell dec_gnd_begin_L)
middle = (set_cell dec_gnd_middle_L)
end = (set_cell dec_gnd_end_L)
longalone1 = (set_cell dec_tran_longalone1_L)
longalone2 = (set_cell dec_tran_longalone2_L)
long = (set_cell dec_tran_longpoly_L)
sht = (set_cell dec_tran_shtpoly_L)
long_middle1 = (set_cell dec_tran_longpoly_middle1_L)
long_middle2 = (set_cell dec_tran_longpoly_middle2_L)
alone = (set_cell dec_tran_alone_L)
dc1 = (set_cell dec_2_L)
dc2 = (set_cell dec_22_L)
notran = (set_cell dec_notran_L)
```

Start a new layout cell

```
set_cell_id (add_cell @LAYOUT dec)
```

Initialization of variables

```
top = 0
lst = ' ()
sav = ' ()
compare = ' ()
a = 1
b = 0
incr = 1
x = 0
count2 = 0
count = 0
for(k = 0; @k < @in; ++k) { incr *= 2 }
save_inc = @incr
for(k=0; @k < (@in - 1); ++k) {
append compare (@save_inc / 4)
save_inc /= 2
}
```

Place the beginning, middle or end transistors

```
for(row = 1; @row <= (@rows / 2); row++) {
println @row
if(@row == 1) {
```

```
append lst @begin
}
if(@row == (@rows / 2)) {
append lst @end
}
if(@row > 1) {
if(@row < (@rows / 2)) {
append lst @middle
}
}
tmp = @count
```

Create the address table

```
sav = '()
for(col = 0; @col < (@in - 1); col++) {

if((@tmp % 2) == 1) {
prepend sav @a
} else { prepend sav @b }
tmp = (trunc(@tmp / 2))
}
```

Place the decoder transistors

```
count += 1
 for(i = 0; @i < (@in - 1); i++) {
if(@sav[@i] == 0) {
if((@i % 2) == 1) {
if((@row % @compare[@i]) == 0) {
if(@compare[@i] == 1) {
append lst @alone
append lst @notran
} else {
append lst @longalone1
append lst @notran }
} else {
if(((@compare[@i] - @row - 2) % 4) == 0) {
append lst @long_middle2
append lst @notran
} else {
if(@row == 1) {
append lst @sht
append lst @notran
} else {
append lst @long
```

```
append lst @notran
}
}
}
} else {
if((@row % @compare[@i]) == 0) {
if(@compare[@i] == 1) {
append lst @notran
append lst @alone
} else {
append lst @notran
append lst @longalone1}
} else {
if(((@compare[@i] - @row - 2) % 4) == 0) {
append lst @notran
append lst @long_middle1
} else {
if(@row == 1) {
append lst @notran
append lst @sht
} else {
append lst @notran
append lst @long
}
}
}
}
} else {
if((@i % 2) == 1) {
if((@row % @compare[@i])== 0) {
if(@compare[@i] == 1) {
append lst @notran
append lst @alone
} else {
append lst @notran
append lst @longalone2 }
} else {
if(((@compare[@i] - @row - 2) % 4) == 0) {
append lst @notran
append lst @long_middle1
} else {
if(@row == 1) {
append lst @notran
append lst @sht
} else {
append lst @notran
```

```
append lst @long
}
}
}
} else {
if((@row % @compare[@i])== 0) {
if(@compare[@i] == 1) {
append lst @alone
append lst @notran
} else {
append lst @longalone2
append lst @notran }
} else {
if(((@compare[@i] - @row - 2) % 4) == 0) {
append lst @long_middle2
append lst @notran
} else {
if(@row == 1) {
append lst @sht
append lst @notran
} else {
append lst @long
append lst @notran
}
}
}
}

}
}
if((@count % 2) == 0) {
append lst @dc2
} else { append lst @dc1 }
}
```

Create the layout cell

```
if(@flip == 0) {
zz = (add_array @lst ((@rows / 2) + 0) ((@in * 2)) (bld @xx @yy) @R0)
} else {
zz = (add_array @lst ((@incr / 2) + 0) ((@in * 2)) (bld @xx @yy) @RX) }
```

Place top level terminals

```
count = 0
source = (make_target out2 @zz ((@rows / 2) - 1) 0)
```

```
source2 = (make_target out0 @zz ((@rows / 2) - 1) ((@in * 2) - 1))
t = (add_terminal (get_named_type GND) (get_named_type MET1) (get_loc
@source) @R0 0 @source)
t = (add_terminal (get_named_type VDD) (get_named_type MET1) (get_loc
@source2) @R0 0 @source2)

source = (make_target in5 @zz 0 ((@in * 2) - 1))
aa = "clk_dec"
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @aa)

count = 1
for(i = 1; @i < ((@in * 2) - 1); i++) {
aa = (cat "in[" @i "]")
source = (make_target out7 @zz ((@rows / 2) - 1) @i)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @aa)
}
count = ((@in * 2) - 1)
aa = (cat "in[" @count "]")
count = (@in * 2)
bb = (cat "in[" @count "]")
source = (make_target out6 @zz ((@rows / 2) - 1) ((@in * 2) - 1))
source2 = (make_target out4 @zz ((@rows / 2) - 1) ((@in * 2) - 1))
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @aa)
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source2) @R0 0 @source2 @bb)

count = 1
for(i = 0; @i < (@rows / 2); i++) {
aa = (cat "out[" @count "]")
count += 1
bb = (cat "out[" @count "]")
count += 1
source = (make_target out2 @zz @i ((@in * 2) - 1))
source2 = (make_target out1 @zz @i ((@in * 2) - 1))
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source) @R0 0 @source @aa)
t = (add_terminal (get_named_type OUT) (get_named_type MET2) (get_loc
@source2) @R0 0 @source2 @bb)
}

}
```

A.8 ROM generator

Beginning of Program

```
func romgen(int word_length numb_blocks rows){
```

Declarations

```
int columns numb_bufs dec_in z i dummy j
int bufcol flip count count2 cnt strap
string aa bb cc dd
float x y
list buf lst1 lst2 lst3
strap = 8
buf = '()
lst1 = '()
lst2 = '()
lst3 = '()
flip = 0
dec_in = 0
abstract Lid buffer z buffer2 tog zz zzz
abstract Lid source sinc bk bk2 bk3 t source2
lread newrowdecbuf_L
```

Load in the generators

```
load read_dec.m
load read_rom.m
load rowdec.m
load rom3.m
```

Assign a cell to an abstract Lid

```
buffer = (set_cell newrowdecbuf_L)
columns = (@word_length * @numb_blocks)
numb_bufs = (@rows / 4)
dummy = 1
```

Calculate certain parameters

```
for(i = 0; @dummy < @rows; i++) {
dummy *= 2
println @dummy
}
println "done"
for(i = (@dummy * 2); (@i - 2) != 0; (i /= 2)){
dec_in += 1
}
println @dec_in
for(i = 0; @i < @numb_bufs; i++) {
append buf @buffer
}
```

Read in the cells for the row decoder and the ROM array

```
read_dec
read_rom

bufcol = 1
x = 0.0
y = 0.0
println @dec_in
```

Call the row decoder generator

```
dc @dec_in @rows @flip @x @y
zz = (set_cell dec)
flip = 1
```

Call the ROM array generator

```
rom @rows @columns @word_length @numb_blocks

zzz = (set_cell rom1)
```

Start a new layout cell

```
set_cell_id (add_cell @LAYOUT romgen)
```

Assign the decoder to an instance and do some boundary calculations

```
bk = (add_instance @zz '(0 0) @R0 0)
lst1 = (get_bbox 0)
println @lst1
```

```
lst2 = @lst1[1]
println @lst2
x = @lst2[0]
y = 5.4
```

Assign the row buffers to an Lid

```
z = (add_array @buf (@numb_bufs) (@bufcol) (bld @x @y) @R0)
lst1 = (get_bbox 0)
println @lst1
lst2 = @lst1[1]
println @lst2
x = @lst2[0]
println @x
y += 5.9
```

Assign the ROM array to an Lid

```
bk3 = (add_instance @zzz (bld @x @y) @R0 0)
count = 0
count2 = 0
```

Wire the different components together and add top level connectors

```
for(i = 0; @i < (@rows / 4); i ++) {
count += 1
aa = (cat "out[" @count "]")
source = (make_target @aa @bk)
sinc = (make_target in3 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
count += 1
aa = (cat "out[" @count "]")
source = (make_target @aa @bk)
sinc = (make_target in1 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
count += 1
aa = (cat "out[" @count "]")
source = (make_target @aa @bk)
sinc = (make_target in2 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
count += 1
aa = (cat "out[" @count "]")
source = (make_target @aa @bk)
```

```
sinc = (make_target in4 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
}
count = 0
count2 = 0
for(i = 0; @i < (@rows / 4); i ++) {
count += 1
aa = (cat "rom_in[" @count "]")
source = (make_target @aa @bk3)
sinc = (make_target out1 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
count += 1
aa = (cat "rom_in[" @count "]")
source = (make_target @aa @bk3)
sinc = (make_target out3 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
count += 1
aa = (cat "rom_in[" @count "]")
source = (make_target @aa @bk3)
sinc = (make_target out4 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
count += 1
aa = (cat "rom_in[" @count "]")
source = (make_target @aa @bk3)
sinc = (make_target out2 @z @i 0)
add_wire @source @sinc (get_named_type MET2) (bld (make_seg \
@HOR 0.0)(make_seg @VER 0.0)(make_seg @HOR 0.0))
}

source = (make_target clk_dec @bk)
aa = "clk_d"
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @aa)
source = (make_target rom_clk @bk3)
aa = "clk_rom"
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @aa)
source = (make_target gnd0 @bk)
t = (add_terminal (get_named_type GND) (get_named_type MET1) (get_loc
@source) @R0 0 @source)
source = (make_target gnd0 @bk3)
t = (add_terminal (get_named_type GND) (get_named_type MET2) (get_loc
```

```
@source) @R0 0 @source)
source = (make_target vdd0 @bk)
aa = "clk_d"
t = (add_terminal (get_named_type VDD) (get_named_type MET1) (get_loc
@source) @R0 0 @source)
source = (make_target vdd0 @bk3)
t = (add_terminal (get_named_type VDD) (get_named_type MET2) (get_loc
@source) @R0 0 @source)
source = (make_target vdd1 @bk3)
t = (add_terminal (get_named_type VDD) (get_named_type MET2) (get_loc
@source) @R0 0 @source)
source = (make_target gnd2 @z 0 0)
aa = "clk_d"
t = (add_terminal (get_named_type GND) (get_named_type MET1) (get_loc
@source) @R0 0 @source)
source = (make_target vdd0 @z 0 0)
t = (add_terminal (get_named_type VDD) (get_named_type MET1) (get_loc
@source) @R0 0 @source)
source = (make_target gnd0 @z 0 0)
t = (add_terminal (get_named_type GND) (get_named_type MET1) (get_loc
@source) @R0 0 @source)
@dec_in
for(i = 1; @i < ((@dec_in * 2) - 1); i++) {
aa = (cat "in[" @i "]")
bb = (cat "in_dec[" @i "]")
source = (make_target @aa @bk)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @bb)
}
count = ((@dec_in * 2) - 1)
aa = (cat "in[" @count "]")
cc = (cat "in_dec[" @count "]")
count = (@dec_in * 2)
bb = (cat "in[" @count "]")
dd = (cat "in_dec[" @count "]")
source = (make_target @aa @bk)
source2 = (make_target @bb @bk)
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @cc)
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source2) @R0 0 @source2 @dd)

count2 = 1
count = 1
cnt = 1
for(j = 0; @j < @numb_blocks; j++) {
```

```
for(i = 0; @i < (@word_length); i++) {
aa = (cat "rom_out[" @count2 "]")
bb = (cat "out_rom[" @count2 "]")
source = (make_target @aa @bk3)
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @bb)
if((@cnt % @strap) == 0) {
count += 1
}
cnt += 1
count += 1
count2 += 1
}
}
count2 = 1
aa = (cat "coldec_in[" @count2 "]")
bb = (cat "col_decin[" @count2 "]")
source = (make_target @aa @bk3)
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @bb)

count = 1
for(i = 1; @i < @numb_blocks; i++) {
count2 = (@count + 1)
aa = (cat "coldec_in[" @count2 "]")
bb = (cat "col_decin[" @count2 "]")
source = (make_target @aa @bk3)
t = (add_terminal (get_named_type IN) (get_named_type POLY) (get_loc
@source) @R0 0 @source @bb)
count += 1
 }

}
```

A.9 Serial Access Memory Generator Code

This is the Lx code for the layout of the serial access memory.
User specifications are:
1.) rows = n
2.) columns = m
3.) name of the SAM cell

```
/*****************************************************************************/
```

Beginning of Program

```
func smem ((int rows cols)(string name)) {
```

Declarations

```
string b a c d e
int i j count k check div
list lst
abstract Lid sm1 sm2 sm3 ad source sinc t source2 sinc sinc2
abstract Lid source3 source4 source5
abstract Lid a1 a2 a3
```

Initialization of variables and assignment of Lids to layout cells

```
check = 0
count = 0
lread and1.L
lread and2.L
lread and3.L
lread smem1.L
lread smem2.L
lread smem3.L
lst = '()
a1 = (set_cell and1)
a2 = (set_cell and2)
a3 = (set_cell and3)
sm1 = (set_cell smem1)
sm2 = (set_cell smem2)
sm3 = (set_cell smem3)
```

Start a new layout cell called name

```
set_cell_id (add_cell @LAYOUT @name)

for(j = 0; @j < @rows; j++) {
      if((@j % 2) == 0) {
append lst @a1
} else {
if(@j == (@rows - 1)) {
append lst @a3
} else { append lst @a2 }
}
for(i = 0; @i < @cols; i++) {
if((@j % 2) == 0) {
append lst @sm1
```

```
} else {
if(@j == (@rows - 1)) {
append lst @sm3
} else { append lst @sm2 }
}
}
}
```

Create the layout

```
ad = (add_array @lst @rows (@cols + 1))
```

Add input and output top level terminals

```
for(i = 0; @i < @rows; i ++) {
a = (cat "din[" @i "]")
b = (cat "dout[" @i "]")
source = (make_target data_in @ad @i 0)
source2 = (make_target data_out @ad @i (@cols))
t = (add_terminal (get_named_type IN) (get_named_type MET1) (get_loc
@source) @R0 0 @source @a)
t = (add_terminal (get_named_type OUT) (get_named_type MET1) (get_loc
@source2) @R0 0 @source2 @b)
}
```

Add Vdd and GND top level terminals

```
source = (make_target in3 @ad 0 1)
source2 = (make_target in2 @ad 0 1)
t = (add_terminal (get_named_type VDD) (get_named_type MET3) (get_loc
@source) @R0 0 @source)
t = (add_terminal (get_named_type GND) (get_named_type MET3) (get_loc
@source2) @R0 0 @source2)

a = (cat "res")
b = (cat "clk")
source = (make_target res @ad (@rows - 1) 0)
source2 = (make_target clk @ad (@rows - 1) 1)
t = (add_terminal (get_named_type IN) (get_named_type MET3) (get_loc
@source) @R0 0 @source @a)
t = (add_terminal (get_named_type IN) (get_named_type MET3) (get_loc
@source2) @R0 0 @source2 @b)
}
```

Appendix B MicroRoute Tips

B.0 Introduction

MicroRoute is a very powerful tool created by Mentor Graphics. It will take a netlist and a block list of a design and wire everything automatically. This appendix is filled with hints on what to do to get a design ready to be used by M.R.

B.1 Steps Before MicroRoute

1.) Use Led to create the layout cells needed for the component.

2.) The input to MicroRoute is the bounding box information of the layouts (the outer boundaries of each of the metal layers for routing purposes). In order to obtain this, run the following program on each of your layout cells:

Lc -B layout_cell.L > layout_cell.LL

The ">" will re-direct the output into the file layout_cell.LL. If you do not do this, Lc will
over write your cell with bounding box information and you will loose all layout information.

3.) Enter Led and create a new Schematic cell. Call all of the .LL files in as instances. Wire the entire layout by hand with schematic wires. This is used to create a netlist for

MicroRoute. Input, output, GND and Vdd terminals should also be placed as top level connectors. These connectors will be used to connect other MicroRouted components to this one. Ofcourse the connectors will be in the schematic level which will need to be changed later on.

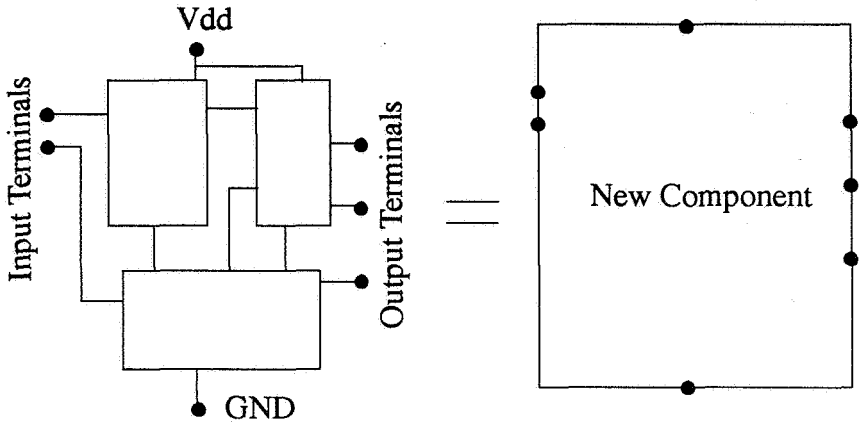


Fig. B.1 Placement of Top Level Connectors

Once you are finished wiring, save the layout as layout.LL for example.

4.) Convert the layout.LL file into a file that MicroRoute can read:

LtoUPR -A mr -t cmos26 -B block.upr -N net.upr -c layout.LL

Now a block file and a net file is all that is needed to start MicroRoute. After this step is accomplished with no errors, then you need to edit your block file. Because MicroRoute does not understand the schematic level input and output terminals, you must edit the
block file to change the input and output terminals that you placed in the schematic wiring session. Do a search for LEV. You will see a line similar to the following:

```
Port p9 (19.000, -319.000) LEV kind=in width=0.2
```

You need to replace all LEV terminals with one of the metal layers such as MET1, MET2, MET3 or POLY. You also need to make the width = 1.0. After your changes, the line should look something like this:

```
Port p9 (19.000, -319.000) MET1 kind=in width=1.0
```

5.) Invoke MicroRoute:

MicroRoute -t cmos26 -B block.upr -N net.upr

6.) The MicroRoute manual is very good from this point on. During the set up phase, here are a few things to change when setting up your Routing:

a.)Tracks and Branches = 5

b.)Channel Exit Margins = 500

c.)Add to the Routing Order the following metal and contacts in order:

MET1

M1M2

MET2

M2M3

MET3

d.)Select MET2 and edit it. Toggle the horizontal to vertical and vertical to horizontal. This will make the routing scheme HVH (MET1 orizontal, MET2 vertical, MET3 horizontal).

e.)Select MET3 and edit it. Change the cost of this layer to be higher: 8/10 instead of 4/6.

f.)Select M2M3 contact. Make the size 3.2 instead of 2.4.

7.) Place the components as close to where they should be before routing (This comes with experience).

8.) Refer to manual from here on.

Appendix C VHDL Code

C.0 Contents

This appendix contains all of the VHDL code to describe the register level components.

C.1 Adder Cell Code

```
LIBRARY unix;
USE unix.math.ALL;
LIBRARY lsim;
USE lsim.terminals.ALL;
USE lsim.pragmas.ALL;
LIBRARY std;
USE std.textio.ALL;

ENTITY adder_cell IS
PORT(a, b, cin : IN LSIM_LOGIC; sout, cout : OUT LSIM_LOGIC);
END adder_cell;

ARCHITECTURE bhv OF adder_cell IS
signal kk : integer;
BEGIN

PROCESS(a, b, cin)

PROCEDURE lsm2int (lsm : IN LSIM_LOGIC_VECTOR; int : OUT INTEGER) IS
variable result : INTEGER;
begin
result := 0;
FOR i in 0 to lsm'LENGTH-1 LOOP
IF lsm(i) = '1' THEN
result := result + 2**i;
END IF;
END LOOP;
int := result;
END lsm2int;

FUNCTION sum (aa, bb, cc : LSIM_LOGIC) RETURN LSIM_LOGIC IS
```

```
BEGIN
RETURN (aa AND bb AND cc) OR (aa AND (NOT bb) AND (NOT cc)) OR ((NOT
aa) AND bb AND (NOT cc)) OR ((NOT aa) AND (NOT bb) AND cc);
END sum;

FUNCTION carry (aa, bb, cc : LSIM_LOGIC) RETURN LSIM_LOGIC IS
BEGIN
RETURN (aa AND bb) OR (aa AND cc) OR (bb AND cc);
END carry;

FILE del_sum : TEXT IS IN "sum.dat";
TYPE DEL IS ARRAY (0 to 63) of time;
variable tmp : DEL;
variable aaa, bbb, ccc: LSIM_LOGIC := '0';
variable x : bit := '1';
variable k, int1, int2, int3 : integer range 0 to 63 := 0;
variable line1, line2 : LINE;
variable abc1, abc2 : LSIM_LOGIC_VECTOR (2 downto 0) ;


BEGIN

if x = '1' THEN
for i in 0 to 63 loop
READLINE(del_sum, line1);
READ(line1, tmp(i));
end loop;
x := '0';
end if;
k := 2;
abc1(k) := ccc;
k := 1;
abc1(k) := bbb;
k := 0;
abc1(k) := aaa;

k := 2;
abc2(k) := cin;
k := 1;
abc2(k) := b;
k := 0;
abc2(k) := a;
lsm2int(abc1, int1);
lsm2int(abc2, int2);
int3 := int1 + (int2 * 8);
kk <= int3;
```

```
sout <= transport sum (a, b, cin) AFTER tmp(int3);
cout <= transport carry (a, b, cin) AFTER tmp(int3);
aaa := a;
bbb := b;
ccc := cin;

END PROCESS;

END bhv;
```

C.2 AND Gate Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY and1 IS
PORT(a, b : IN LSIM_LOGIC; c : OUT LSIM_LOGIC);
END and1;

ARCHITECTURE bhv OF and1 IS

FUNCTION andab (aa, bb : LSIM_LOGIC) RETURN LSIM_LOGIC IS
BEGIN
RETURN (aa AND bb);
END andab;

BEGIN

c <= andab (a, b);

END bhv;
```

C.3 Counter Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY count16 IS
PORT (clk, reset : IN LSIM_LOGIC; cnt1, cnt2, cnt3, cnt4 : OUT LSIM_LOGIC);
END count16;

ARCHITECTURE bhv OF count16 IS

BEGIN

PROCESS (clk)
```

```
variable count : INTEGER RANGE 0 to 16 := 15;

BEGIN

IF clk = '1' THEN
IF reset = '0' THEN
count := count + 1;
ELSE
count := 0;
END IF;

IF count > 15 THEN
count := 0;
END IF;

IF count = 0 THEN
cnt1 <= '0';
cnt2 <= '0';
cnt3 <= '0';
cnt4 <= '0';
ELSIF count = 1 THEN
cnt1 <= '0';
cnt2 <= '0';
cnt3 <= '0';
cnt4 <= '1';
ELSIF count = 2 THEN
cnt1 <= '0';
cnt2 <= '0';
cnt3 <= '1';
cnt4 <= '0';
ELSIF count = 3 THEN
cnt1 <= '0';
cnt2 <= '0';
cnt3 <= '1';
cnt4 <= '1';
ELSIF count = 4 THEN
cnt1 <= '0';
cnt2 <= '1';
cnt3 <= '0';
cnt4 <= '0';
ELSIF count = 5 THEN
cnt1 <= '0';
cnt2 <= '1';
cnt3 <= '0';
cnt4 <= '1';
```

```
ELSIF count = 6 THEN
cnt1 <= '0';
cnt2 <= '1';
cnt3 <= '1';
cnt4 <= '0';
ELSIF count = 7 THEN
cnt1 <= '0';
cnt2 <= '1';
cnt3 <= '1';
cnt4 <= '1';
ELSIF count = 8 THEN
cnt1 <= '1';
cnt2 <= '0';
cnt3 <= '0';
cnt4 <= '0';
ELSIF count = 9 THEN
cnt1 <= '1';
cnt2 <= '0';
cnt3 <= '0';
cnt4 <= '1';
ELSIF count = 10 THEN
cnt1 <= '1';
cnt2 <= '0';
cnt3 <= '1';
cnt4 <= '0';
ELSIF count = 11 THEN
cnt1 <= '1';
cnt2 <= '0';
cnt3 <= '1';
cnt4 <= '1';
ELSIF count = 12 THEN
cnt1 <= '1';
cnt2 <= '1';
cnt3 <= '0';
cnt4 <= '0';
ELSIF count = 13 THEN
cnt1 <= '1';
cnt2 <= '1';
cnt3 <= '0';
cnt4 <= '1';
ELSIF count = 14 THEN
cnt1 <= '1';
cnt2 <= '1';
cnt3 <= '1';
cnt4 <= '0';
ELSIF count = 15 THEN
```

```
cnt1 <= '1';
cnt2 <= '1';
cnt3 <= '1';
cnt4 <= '1';
END IF;

END IF;

END PROCESS;

END bhv;
```

C.4 Inverter Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY inv IS
PORT(a : IN LSIM_LOGIC; ab : OUT LSIM_LOGIC);
END inv;

ARCHITECTURE bhv OF inv IS

FUNCTION inv2 (aa : LSIM_LOGIC) RETURN LSIM_LOGIC IS
BEGIN
RETURN (NOT aa);
END inv2;

BEGIN

ab <= inv2 (a);

END bhv;
```

C.5 Latch Cell Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY latch IS
PORT (reset, clk, a : IN LSIM_LOGIC; b : OUT LSIM_LOGIC);
END latch;

ARCHITECTURE bhv OF latch IS

BEGIN
```

```
PROCESS (clk)

variable save : LSIM_LOGIC := '0';

BEGIN

IF clk = '1' THEN
IF reset = '0' THEN
b <= '0';
ELSE
b <= a;
save := a;
END IF;
ELSE

IF reset = '0' THEN
b <= '0';
ELSE
b <= save;
END IF;
END IF;

END PROCESS;

END bhv;
```

C.6 Master Section of the SAM Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY master IS
PORT (reset, clk, a : IN LSIM_LOGIC; b : OUT LSIM_LOGIC);
END master;

ARCHITECTURE bhv OF master IS

BEGIN

PROCESS (clk, a)

variable save : LSIM_LOGIC := '0';

BEGIN
```

```
IF clk = '1' THEN
IF reset = '0' THEN
b <= '0';
ELSE
b <= a;
save := a;
END IF;
ELSE

IF reset = '0' THEN
b <= '0';
ELSE
b <= save;
END IF;
END IF;

END PROCESS;

END bhv;
```

C.7 Multiplexer Cell Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY mux IS
PORT(a, b, carry, clk : IN LSIM_LOGIC; c : OUT LSIM_LOGIC);
END mux;

ARCHITECTURE bhv OF mux IS

BEGIN

PROCESS (clk)

BEGIN

IF carry = '1' THEN
c <= a;
ELSE
c <= b;
END IF;

END PROCESS;

END bhv;
```

C.8 One Shot Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY one_shot IS
PORT(clk, a : IN LSIM_LOGIC; b : OUT LSIM_LOGIC);
END one_shot;

ARCHITECTURE bhv OF one_shot IS

BEGIN

PROCESS (clk)

variable out1 :LSIM_LOGIC := '0';
variable save, count : LSIM_LOGIC := '0';

BEGIN

IF clk = '1' THEN

IF (a = '1') AND (save = '1') AND (count = '1') THEN
out1 := '1';
END IF;
IF (a = '1') AND (save = '0') THEN
out1 := '0';
save := '1';
END IF;
IF (a = '0') AND (save = '1') AND (count = '0') THEN
count := '1';
END IF;
END IF;

b <= out1;

END PROCESS;

END bhv;
```

C.9 OR Cell Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;
```

```
ENTITY or2 IS
PORT(a, b : IN LSIM_LOGIC; c : OUT LSIM_LOGIC);
END or2;

ARCHITECTURE bhv OF or2 IS

FUNCTION orab (aa, bb : LSIM_LOGIC) RETURN LSIM_LOGIC IS
BEGIN
RETURN (aa OR bb);
END orab;

BEGIN

c <= orab (a, b);

END bhv;
```

C.10 Quadrant Detection Unit Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY qd_unit IS
PORT (clk, a, b, c, d, i_in, q_in : IN LSIM_LOGIC; i_out, q_out: OUT
LSIM_LOGIC);
END qd_unit;

ARCHITECTURE bhv OF qd_unit IS

BEGIN

PROCESS (clk, a, b, c, d, i_in, q_in)

variable save : LSIM_LOGIC := '0';

BEGIN

IF ((a = '1') OR (b = '1') OR (c = '1') OR (d = '1')) THEN

IF (a = '1') THEN
i_out <= i_in;
q_out <= q_in;
END IF;
IF (b = '1') THEN
i_out <= (NOT q_in);
q_out <= i_in;
```

```
END IF;
IF (c = '1') THEN
i_out <= q_in;
q_out <= (NOT i_in);
END IF;
IF (d = '1') THEN
i_out <= (NOT i_in);
q_out <= (NOT q_in);
END IF;

ELSE
i_out <= i_in;
q_out <= q_in;
END IF;

END PROCESS;

END bhv;
```

C.11 Slave Section of the SAM Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY slave IS
PORT (clk, a : IN LSIM_LOGIC; b : OUT LSIM_LOGIC);
END slave;

ARCHITECTURE bhv OF slave IS

BEGIN

PROCESS (clk)

variable save : LSIM_LOGIC := '0';

BEGIN

IF clk = '1' THEN
                b <= a;
                save := a;
ELSE

                b <= save;
END IF;
```

```
END PROCESS;

END bhv;
```

C.12 XOR Cell Code

```
LIBRARY lsim;
USE lsim.terminals.ALL;

ENTITY xor21 IS
PORT(a, sign : IN LSIM_LOGIC; xout : OUT LSIM_LOGIC);
END xor21;

ARCHITECTURE bhv OF xor21 IS

FUNCTION xorab (aa, bb : LSIM_LOGIC) RETURN LSIM_LOGIC IS
BEGIN
RETURN (aa XOR bb);
END xorab;

BEGIN

xout <= xorab (a, sign);

END bhv;
```

C.13 Modulator Samples Code

```
LIBRARY unix;
USE unix.math.ALL;
LIBRARY lsim;
USE lsim.terminals.ALL;
USE lsim.pragmas.ALL;

ENTITY mod_samp IS
PORT (clk, en : IN LSIM_LOGIC; mod0, mod1, mod2, mod3, mod4, mod5, mod6,
mod7, rndi, rndq : OUT LSIM_LOGIC);
END mod_samp;

ARCHITECTURE bhv OF mod_samp IS

--signal seed : REAL := 1951.0;
signal rnd_number : REAL := 0.0;
signal fourth : real := 0.0;
signal symbol : real := 0.0;
signal twt_out : real := 0.0;
```

```
signal modul : real := 0.0;
signal ans : real := 0.0;
signal ffff : real := 0.0;

BEGIN

PROCESS (clk)

PROCEDURE frac2lsm(good : IN real; lsm : OUT LSIM_LOGIC_VECTOR; out1 :
OUT real) IS
variable tmp1, tmp2 : real;
variable iii : integer;
variable f : real;
variable check : LSIM_LOGIC_VECTOR (7 downto 0);

BEGIN

iii := 7;
f := good;
IF (f >= 0.0) THEN
check(iii) := '0';
ELSE
check(iii) := '1';
f := (f * (-1.0));
f := 1.0 - f;
END IF;
for iii in 6 downto 0 LOOP
tmp1 := 2.0 * f;
tmp2 := floor(tmp1);
out1 := tmp2;
f := tmp1 - tmp2;
IF tmp2 = 1.0 THEN
check(iii) := '1';
ELSE
check(iii) := '0';
END IF;
END LOOP;

lsm := check;
END frac2lsm;

PROCEDURE random(seed_in : IN REAL; seed_g, rand_g, sed : OUT REAL; id,
qd : OUT LSIM_LOGIC) IS

variable k : REAL := 65539.0;
variable m : REAL := 2147483648.0;
```

```
variable sd, rd : REAL := 0.0;

BEGIN

sd := seed_in;
sd := remainder((k * sd), m);
sd := sd / m;
IF sd < 0.0 THEN
sd := 1.0 + sd;
END IF;
sed := sd;
seed_g := sd * m;

rd := (sd * m) / (m - 1.0);

IF((rd >= 0.0) AND (rd < 0.25)) THEN
rand_g := 1.0;
id := '1';
qd := '0';
END IF;

IF((rd >= 0.25) AND (rd < 0.5)) THEN
rand_g := 2.0;
id := '0';
qd := '0';
END IF;

IF((rd >= 0.5) AND (rd < 0.75)) THEN
rand_g := 3.0;
id := '0';
qd := '1';
END IF;

IF((rd >= 0.75) AND (rd <= 1.0)) THEN
rand_g := 4.0;
id := '1';
qd := '1';
END IF;
END random;

PROCEDURE bv2lsmv (bin : IN BIT; lsm : OUT LSIM_LOGIC) IS
BEGIN
IF bin = '1' THEN
lsm := '1';
ELSE
lsm := '0';
```

```
END IF;
END bv2lsmv;

PROCEDURE modulate(sym, n : IN REAL; i_wave, q_wave : OUT REAL) IS
variable fc : REAL := 25000000.0;
variable T : REAL := 1.0 / 100000000.0;
variable Es : REAL := 0.5;
variable pi : REAL := 3.141592654;
variable theta : REAL := 1.9;

BEGIN

i_wave := (sqrt(2.0 * Es) * cos((2.0 * pi * fc * n * T ) + (((2.0 * sym) - 1.0) * (pi / 4.0))
+ theta));
q_wave := (sqrt(2.0 * Es) * sin((2.0 * pi * fc * n * T ) + (((2.0 * sym) - 1.0) * (pi / 4.0))
+ theta));
END modulate;

TYPE arr IS ARRAY (0 to 14) of BIT;
TYPE arr2 IS ARRAY (0 to 14) of real;
variable i, j: INTEGER := 0;
variable mod_samp : INTEGER;
variable seed, seed_gen : real := 1951.0;
variable modi_out, modq_out : real;
variable time : real := 0.0;
variable ii : real;
variable ioutt, qoutt : BIT;
variable iout2, qout2 : LSIM_LOGIC;
variable iout : arr := ('0', '1', '1', '1', '1', '0', '0', '0', '1', '0', '0', '1', '0', '1', '0');
variable qout : arr := ('0', '1', '1', '1', '1', '0', '0', '0', '1', '0', '0', '1', '0', '1', '0');
variable symb : arr2 := (2.0, 4.0, 4.0, 4.0, 4.0, 2.0, 2.0, 2.0, 4.0, 2.0, 2.0, 4.0,
2.0, 4.0, 2.0);
variable seed_save : real;
variable xx : LSIM_LOGIC := '1';
variable iiii, qqqq : LSIM_LOGIC;
variable rnd_num, s, rnd_save, symb_save : real;
variable mod_out : real := 0.0;
variable modlsm : LSIM_LOGIC_VECTOR (7 downto 0);
variable four : real := 0.0;
variable tmperary : integer := -1;
variable a1, b1, c1, d1, e1, f1, g1, h1, i1 : real := 0.0;
variable j1, k1, l1, m1, n1, o1 : real := 0.0;
variable a2, b2, c2, d2, e2, f2, g2, h2, i2 : real := 0.0;
variable j2, k2, l2, m2, n2, o2 : real := 0.0;
variable pp : integer := 0;
variable fff : real;
```

```
BEGIN

IF en = '1' THEN
IF clk = '1' THEN
-- Phase Lock

IF time < 64.0 THEN
ii := 4.0;
symbol <= ii;
rndi <= '1';
rndq <= '1';
modulate(ii, time, modi_out, modq_out);
modul <= modi_out;
twt_out <= modi_out;
frac2lsm(modi_out, modlsm,fff);
ffff <= fff;
pp := 0;
mod0 <= modlsm(pp);
pp := 1;
mod1 <= modlsm(pp);
pp := 2;
mod2 <= modlsm(pp);
pp := 3;
mod3 <= modlsm(pp);
pp := 4;
mod4 <= modlsm(pp);
pp := 5;
mod5 <= modlsm(pp);
pp := 6;
mod6 <= modlsm(pp);
pp := 7;
mod7 <= modlsm(pp);
IF (time = 63.0) THEN
ii := 2.0;
END IF;
END IF;

-- Timing Recovery
IF ((time >= 64.0) AND (time < 143.0)) THEN
IF four = 0.0 THEN
IF ii = 2.0 THEN
ii := 4.0;
symbol <= ii;
rndi <= '1';
rndq <= '1';
```

```
ELSE
ii := 2.0;
symbol <= ii;
rndi <= '0';
rndq <= '0';
END IF;
END IF;
four := four + 1.0;
fourth <= four + 1.0;
IF four = 4.0 THEN
four := 0.0;
fourth <= 0.0;
END IF;
modulate(ii, time, modi_out, modq_out);
modul <= modi_out;
frac2lsm(modi_out, modlsm, fff);
ffff <= fff;
pp := 0;
mod0 <= modlsm(pp);
pp := 1;
mod1 <= modlsm(pp);
pp := 2;
mod2 <= modlsm(pp);
pp := 3;
mod3 <= modlsm(pp);
pp := 4;
mod4 <= modlsm(pp);
pp := 5;
mod5 <= modlsm(pp);
pp := 6;
mod6 <= modlsm(pp);
pp := 7;
mod7 <= modlsm(pp);
END IF;

-- Unique Word
IF ((time >= 143.0) AND (time <= 203.0)) THEN
IF four = 0.0 THEN
tmperary := tmperary + 1;
ioutt := iout(tmperary);
qoutt := qout(tmperary);
bv2lsmv(ioutt, iout2);
bv2lsmv(qoutt, qout2);
rndi <= iout2;
rndq <= qout2;
ii := symb(tmperary);
```

```
symbol <= ii;
END IF;
four := four + 1.0;
fourth <= four + 1.0;
IF four = 4.0 THEN
four := 0.0;
fourth <= 0.0;
END IF;
modulate(ii, time, modi_out, modq_out);
modul <= modi_out;
frac2lsm(modi_out, modlsm, fff);
ffff <= fff;
pp := 0;
mod0 <= modlsm(pp);
pp := 1;
mod1 <= modlsm(pp);
pp := 2;
mod2 <= modlsm(pp);
pp := 3;
mod3 <= modlsm(pp);
pp := 4;
mod4 <= modlsm(pp);
pp := 5;
mod5 <= modlsm(pp);
pp := 6;
mod6 <= modlsm(pp);
pp := 7;
mod7 <= modlsm(pp);
END IF;

-- Data transmission
IF (time >= 204.0) THEN
IF four = 0.0 THEN
IF xx = '0' THEN
seed := seed_gen;
ELSE
seed := 1951.0;
xx := '0';
END IF;
random(seed, seed_gen, rnd_num, s, iiii, qqqq);
rndi <= iiii;
rndq <= qqqq;
END IF;
four := four + 1.0;
fourth <= four + 1.0;
IF four = 4.0 THEN
```

```
four := 0.0;
fourth <= 0.0;
END IF;
rnd_save := s;
rnd_number <= s;
ii := rnd_num;
symbol <= ii;
symb_save := ii;
modulate(ii, time, modi_out, modq_out);
modul <= modi_out;
frac2lsm(modi_out, modlsm, fff);
ffff <= fff;
pp := 0;
mod0 <= modlsm(pp);
pp := 1;
mod1 <= modlsm(pp);
pp := 2;
mod2 <= modlsm(pp);
pp := 3;
mod3 <= modlsm(pp);
pp := 4;
mod4 <= modlsm(pp);
pp := 5;
mod5 <= modlsm(pp);
pp := 6;
mod6 <= modlsm(pp);
pp := 7;
mod7 <= modlsm(pp);
END IF;
time := time + 1.0;
END IF;
END IF;
END PROCESS;

END bhv;
```

C.14 Phase ROM Code

```
LIBRARY unix;
USE unix.math.ALL;
LIBRARY lsim;
USE lsim.terminals.ALL;
USE lsim.pragmas.ALL;

USE std.textio.ALL;
ENTITY phase_rom is
```

```
PORT(clk, i_in1, i_in2, i_in3, i_in4, i_in5, q_in1, q_in2, q_in3, q_in4, q_in5 : IN
LSIM_LOGIC; ph_1, ph_2, ph_3, ph_4, ph_5, ph_6 : OUT LSIM_LOGIC);
END phase_rom;

ARCHITECTURE bhv of phase_rom is

signal cosowt : REAL;

begin

PROCESS (clk)

PROCEDURE lsm2int (lsm : IN LSIM_LOGIC_VECTOR; int : OUT INTEGER) IS
variable result : INTEGER;
begin
result := 0;
FOR i in 0 to lsm'LENGTH-1 LOOP
if lsm(i) = '1' THEN
result := result + 2**i;
END IF;
END LOOP;
int := result;
END lsm2int;

PROCEDURE bin2lsm (bin : IN LSIM_LOGIC_VECTOR; lsm : OUT
LSIM_LOGIC_VECTOR) IS
BEGIN
FOR i in bin'LENGTH-1 downto 0 LOOP
IF bin(bin'LENGTH-1 - i) = '1' THEN
lsm(i) := '1';
ELSE
lsm(i) := '0';
END IF;
END LOOP;
END bin2lsm;

PROCEDURE bv2lsmv (bin : IN BIT_VECTOR; lsm : OUT
LSIM_LOGIC_VECTOR) IS
BEGIN
FOR i in 0 to bin'LENGTH-1 LOOP
IF bin(bin'LENGTH-1 - i) = '1' THEN
lsm(bin'LENGTH-1 - i) := '1';
ELSE
lsm(bin'LENGTH-1 - i) := '0';
END IF;
END LOOP;
```

```
END bv2lsmv;


PROCEDURE signed_bin2frac (bin : IN LSIM_LOGIC_VECTOR ; int : OUT real)
IS
variable result : real;
begin
result := 0.0;
FOR i in 1 to bin'LENGTH-1 LOOP
if bin(bin'LENGTH-1 - i) = '1' THEN
result := result + 2.0**(-i);
END IF;
END LOOP;
int := result;
END signed_bin2frac;

TYPE rom_dat IS FILE OF BIT;
FILE input_cos : rom_dat IS IN "phase.dat";
TYPE MEMORY IS ARRAY (0 to 1023) OF BIT_VECTOR (5 downto 0);
TYPE MEM IS ARRAY (0 to 6143) of BIT;

variable cos1 : LSIM_LOGIC_VECTOR (5 DOWNTO 0);
variable intgr : INTEGER range 0 to 1023 := 0;
variable sav_cos_array : MEMORY;
variable tmp1 : MEM;
variable tmp2 : MEM;
variable x : LSIM_LOGIC :='1';
variable k : integer := 0;
variable sn, cs : real;
variable cosout : LSIM_LOGIC_VECTOR (5 DOWNTO 0);
variable cos_out : LSIM_LOGIC_VECTOR (5 DOWNTO 0);
variable sc_address : LSIM_LOGIC_VECTOR (9 DOWNTO 0);

BEGIN

IF clk = '1' THEN
if x = '1' then

for i in 0 to 6143 loop
READ(input_cos, tmp2(i));
end loop;

FOR i in 0 to 1023 loop
sav_cos_array(i) :=
tmp2(k)&tmp2(k+1)&tmp2(k+2)&tmp2(k+3)&tmp2(k+4)&tmp2(k+5);
k := k+6;
```

```
end loop;
end if;

x := '0';
k := 0;
sc_address(k) := q_in1;
k := 1;
sc_address(k) := q_in2;
k := 2;
sc_address(k) := q_in3;
k := 3;
sc_address(k) := q_in4;
k := 4;
sc_address(k) := q_in5;
k := 5;
sc_address(k) := i_in1;
k := 6;
sc_address(k) := i_in2;
k := 7;
sc_address(k) := i_in3;
k := 8;
sc_address(k) := i_in4;
k := 9;
sc_address(k) := i_in5;

lsm2int(sc_address,intgr);

bv2lsmv(sav_cos_array(intgr), cos1);

signed_bin2frac(cos1, cs);
bin2lsm(cos1, cosout);
cos_out := cosout;
cosowt <= cs;
k := 5;
ph_1 <= cos_out(k);
k := 4;
ph_2 <= cos_out(k);
k := 3;
ph_3 <= cos_out(k);
k := 2;
ph_4 <= cos_out(k);
k := 1;
ph_5 <= cos_out(k);
k := 0;
ph_6 <= cos_out(k);
```

```
END IF;

END PROCESS;

END bhv;
```

C.15 Viterbi Non Linear ROM Code

```
LIBRARY unix;
USE unix.math.ALL;
LIBRARY lsim;
USE lsim.terminals.ALL;
USE lsim.pragmas.ALL;

USE std.textio.ALL;
ENTITY nlin_rom is
PORT(clk, i0, i1, i2, i3, i4, i5, q0, q1, q2, q3, q4, q5 : IN LSIM_LOGIC; io0, io1, io2,
io3, io4, io5, qo0, qo1, qo2, qo3, qo4, qo5 : OUT LSIM_LOGIC);
END nlin_rom;

ARCHITECTURE bhv of nlin_rom is

signal nl_address : LSIM_LOGIC_VECTOR (11 DOWNTO 0);
signal iii : INTEGER;
signal sinowt, cosowt : real;

begin

PROCESS (clk, i0, i1, i2, i3, i4, i5, q0, q1, q2, q3, q4, q5)

PROCEDURE signed_lsm2int (lsm : IN LSIM_LOGIC_VECTOR; int : OUT
INTEGER) IS
variable result : INTEGER;
begin
result := 0;
FOR i in 0 to lsm'LENGTH-1 LOOP
IF lsm(i) = '1' THEN
result := result + 2**i;
END IF;
END LOOP;
int := result;
END signed_lsm2int;

PROCEDURE bin2lsm (bin : IN LSIM_LOGIC_VECTOR; lsm : OUT
LSIM_LOGIC_VECTOR) IS
```

```
BEGIN
FOR i in bin'LENGTH-1 downto 0 LOOP
IF bin(bin'LENGTH-1 - i) = '1' THEN
lsm(i) := '1';
ELSE
lsm(i) := '0';
END IF;
END LOOP;
END bin2lsm;

PROCEDURE bv2lsmv (bin : IN BIT_VECTOR; lsm : OUT
LSIM_LOGIC_VECTOR) IS
BEGIN
FOR i in 0 to bin'LENGTH-1 LOOP
IF bin(bin'LENGTH-1 - i) = '1' THEN
lsm(bin'LENGTH-1 - i) := '1';
ELSE
lsm(bin'LENGTH-1 - i) := '0';
END IF;
END LOOP;
END bv2lsmv;

PROCEDURE signed_bin2frac (bin : IN LSIM_LOGIC_VECTOR ; int : OUT real)
IS
variable result : real;
begin
result := 0.0;
FOR i in 1 to bin'LENGTH-2 LOOP
if bin(bin'LENGTH-1 - i) = '1' THEN
result := result + 2.0**(-i);
END IF;
END LOOP;
IF bin(bin'LENGTH-1) = '1' THEN
int := ((1.0 - result) * (-1.0));
ELSE
int := result;
END IF;
END signed_bin2frac;

TYPE rom_dat IS FILE OF BIT;
FILE input_sin : rom_dat IS IN "vnlq.dat";
FILE input_cos : rom_dat IS IN "vnli.dat";
TYPE MEMORY IS ARRAY (0 to 4095) OF BIT_VECTOR (5 downto 0);
TYPE MEM IS ARRAY (0 to 24575) of BIT;

variable sin1, cos1 : LSIM_LOGIC_VECTOR (5 DOWNTO 0);
```

```
variable sav_sin_array : MEMORY;
variable sav_cos_array : MEMORY;
variable tmp1 : MEM;
variable tmp2 : MEM;
variable x : LSIM_LOGIC :='1';
variable k : integer := 0;
variable sn, cs : real;
variable cosout, sinout : LSIM_LOGIC_VECTOR (5 DOWNTO 0);
variable i_out, q_out : LSIM_LOGIC_VECTOR (5 DOWNTO 0);
variable intgr : integer;
BEGIN

IF clk = '1' THEN
if x = '1' then

for i in 0 to 24575 loop
READ(input_sin, tmp1(i));
READ(input_cos, tmp2(i));
end loop;

FOR i in 0 to 4095 loop
sav_sin_array(i) :=
tmp1(k)&tmp1(k+1)&tmp1(k+2)&tmp1(k+3)&tmp1(k+4)&tmp1(k+5);
sav_cos_array(i) :=
tmp2(k)&tmp2(k+1)&tmp2(k+2)&tmp2(k+3)&tmp2(k+4)&tmp2(k+5);
k := k+6;
end loop;
end if;

x := '0';

k := 0;
nl_address(k) <= q0;
k := 1;
nl_address(k) <= q1;
k := 2;
nl_address(k) <= q2;
k := 3;
nl_address(k) <= q3;
k := 4;
nl_address(k) <= q4;
k := 5;
nl_address(k) <= q5;
k := 6;
nl_address(k) <= i0;
k := 7;
```

```
nl_address(k) <= i1;
k := 8;
nl_address(k) <= i2;
k := 9;
nl_address(k) <= i3;
k := 10;
nl_address(k) <= i4;
k := 11;
nl_address(k) <= i5;

signed_lsm2int(nl_address,intgr);
iii <= intgr;
bv2lsmv(sav_cos_array(intgr), cos1);
bv2lsmv(sav_sin_array(intgr), sin1);

signed_bin2frac(cos1, cs);
signed_bin2frac(sin1, sn);
bin2lsm(cos1, cosout);
bin2lsm(sin1, sinout);
i_out := cosout;
q_out := sinout;

k := 5;
io0 <= i_out(k);
k := 4;
io1 <= i_out(k);
k := 3;
io2 <= i_out(k);
k := 2;
io3 <= i_out(k);
k := 1;
io4 <= i_out(k);
k := 0;
io5 <= i_out(k);

k := 5;
qo0 <= q_out(k);
k := 4;
qo1 <= q_out(k);
k := 3;
qo2 <= q_out(k);
k := 2;
qo3 <= q_out(k);
k := 1;
qo4 <= q_out(k);
k := 0;
```

```
qo5 <= q_out(k);

sinowt <= sn;
cosowt <= cs;

END IF;

END PROCESS;

END bhv;
```

C.16 Numerically Controlled Oscillator ROM Code

```
LIBRARY unix;
USE unix.math.ALL;
LIBRARY lsim;
USE lsim.terminals.ALL;
USE lsim.pragmas.ALL;

USE std.textio.ALL;
ENTITY nco_rom is
PORT(clk, in0, in1, in2, in3, in4, in5, in6, in7, in8, in9 : IN LSIM_LOGIC; c0, c1, c2,
c3, c4, c5, c6, c7, s0, s1, s2, s3, s4, s5, s6, s7 : OUT LSIM_LOGIC);
END nco_rom;

ARCHITECTURE bhv of nco_rom is

signal iii : INTEGER;
signal sinowt, cosowt : real;

begin

PROCESS (clk)

PROCEDURE signed_lsm2int (lsm : IN LSIM_LOGIC_VECTOR; int : OUT
INTEGER) IS
variable result : INTEGER;
begin
result := 0;
FOR i in 0 to lsm'LENGTH-1 LOOP
IF lsm(i) = '1' THEN
result := result + 2**i;
END IF;
END LOOP;
```

```
int := result;
END signed_lsm2int;

PROCEDURE bin2lsm (bin : IN LSIM_LOGIC_VECTOR; lsm : OUT
LSIM_LOGIC_VECTOR) IS
BEGIN
FOR i in bin'LENGTH-1 downto 0 LOOP
IF bin(bin'LENGTH-1 - i) = '1' THEN
lsm(i) := '1';
ELSE
lsm(i) := '0';
END IF;
END LOOP;
END bin2lsm;

PROCEDURE bv2lsmv (bin : IN BIT_VECTOR; lsm : OUT
LSIM_LOGIC_VECTOR) IS
BEGIN
FOR i in 0 to bin'LENGTH-1 LOOP
IF bin(bin'LENGTH-1 - i) = '1' THEN
lsm(bin'LENGTH-1 - i) := '1';
ELSE
lsm(bin'LENGTH-1 - i) := '0';
END IF;
END LOOP;
END bv2lsmv;

PROCEDURE signed_bin2frac (bin : IN LSIM_LOGIC_VECTOR ; int : OUT real)
IS
variable result : real;
begin
result := 0.0;
FOR i in 1 to bin'LENGTH-2 LOOP
if bin(bin'LENGTH-1 - i) = '1' THEN
result := result + 2.0**(-i);
END IF;
END LOOP;
IF bin(bin'LENGTH-1) = '1' THEN
int := ((1.0 - result) * (-1.0));
ELSE
int := result;
END IF;
END signed_bin2frac;

TYPE rom_dat IS FILE OF BIT;
FILE input_cos : rom_dat IS IN "nco_rom.dat";
```

```
TYPE MEMORY IS ARRAY (0 to 2047) OF BIT_VECTOR (7 downto 0);
TYPE MEM IS ARRAY (0 to 16384) of BIT;

variable sin1, cos1 : LSIM_LOGIC_VECTOR (7 DOWNTO 0);
variable sav_sin_array : MEMORY;
variable sav_cos_array : MEMORY;
variable tmp1 : MEM;
variable tmp2 : MEM;
variable x : LSIM_LOGIC :='1';
variable k : integer := 0;
variable sn, cs : real;
variable cosout, sinout : LSIM_LOGIC_VECTOR (7 DOWNTO 0);
variable i_out, q_out : LSIM_LOGIC_VECTOR (7 DOWNTO 0);
variable intgr : integer;
variable nl_address : LSIM_LOGIC_VECTOR (9 DOWNTO 0);
BEGIN

IF clk = '1' THEN
if x = '1' then

for i in 0 to 16383 loop
READ(input_cos, tmp2(i));
end loop;

FOR i in 0 to 1023 loop
sav_cos_array(i) :=
tmp2(k)&tmp2(k+1)&tmp2(k+2)&tmp2(k+3)&tmp2(k+4)&tmp2(k+5)&tmp2(k+6)&t
mp2(k+7);
k := k+8;
sav_sin_array(i) :=
tmp2(k)&tmp2(k+1)&tmp2(k+2)&tmp2(k+3)&tmp2(k+4)&tmp2(k+5)&tmp2(k+6)&t
mp2(k+7);
k := k+8;
end loop;
end if;

x := '0';

k := 0;
nl_address(k) := in0;
k := 1;
nl_address(k) := in1;
k := 2;
nl_address(k) := in2;
k := 3;
nl_address(k) := in3;
```

```
k := 4;
nl_address(k) := in4;
k := 5;
nl_address(k) := in5;
k := 6;
nl_address(k) := in6;
k := 7;
nl_address(k) := in7;
k := 8;
nl_address(k) := in8;
k := 9;
nl_address(k) := in9;

signed_lsm2int(nl_address,intgr);
iii <= intgr;
bv2lsmv(sav_cos_array(intgr), cos1);
bv2lsmv(sav_sin_array(intgr), sin1);

signed_bin2frac(cos1, cs);
signed_bin2frac(sin1, sn);
bin2lsm(cos1, cosout);
bin2lsm(sin1, sinout);
i_out := cosout;
q_out := sinout;

k := 7;
c0 <= i_out(k);
k := 6;
c1 <= i_out(k);
k := 5;
c2 <= i_out(k);
k := 4;
c3 <= i_out(k);
k := 3;
c4 <= i_out(k);
k := 2;
c5 <= i_out(k);
k := 1;
c6 <= i_out(k);
k := 0;
c7 <= i_out(k);

k := 7;
s0 <= q_out(k);
k := 6;
s1 <= q_out(k);
```

```
k := 5;
s2 <= q_out(k);
k := 4;
s3 <= q_out(k);
k := 3;
s4 <= q_out(k);
k := 2;
s5 <= q_out(k);
k := 1;
s6 <= q_out(k);
k := 0;
s7 <= q_out(k);

sinowt <= sn;
cosowt <= cs;

END IF;

END PROCESS;

END bhv;
```

Appendix D Simulation In C

D.0 Modulator Code

```
#include <math.h>
#include <stdio.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr;
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1 + IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0 - EPS)
#define PI 3.141592653589793

/*************************************************/
/* FFT/IFFT:                                     */
/* This subroutine performs the fft on a vector  */
/* when isign is 1 and it performs the ifft when */
/* isign is -1.                                  */
/*************************************************/
void FOUR1(double data[], unsigned long nn, int isign)
{
        unsigned long n, mmax, m, j, istep, i;
        double wtemp, wr, wpr, wpi, wi, theta;
        double tempr, tempi;

        n=nn << 1;
        j=1;
        for(i=1;i<n;i+=2)
        {
```

```
                        if(j > i)
                        {
                                SWAP(data[j],data[i]);
                                SWAP(data[j+1],data[i+1]);
                        }
                        m=n >> 1;
                        while (m >= 2 && j > m)
                        {
                                j -= m;
                                m >>= 1;
                        }
                        j += m;
                }

                mmax = 2;
                while (n > mmax)
                {
                        istep = mmax << 1;
                        theta = isign*(6.28318530717959/mmax);
                        wtemp = sin(0.5 * theta);
                        wpr = -2.0*wtemp*wtemp;
                        wpi = sin(theta);
                        wr = 1.0;
                        wi = 0.0;
                        for(m=1;m<mmax;m+=2)
                        {
                                for(i=m;i<=n;i+=istep)
                                {
                                        j=i+mmax;
                                        tempr = wr * data[j] - wi *
data[j+1];
                                        tempi = wr * data[j+1] + wi *
data[j];
                                        data[j] = data[i]-tempr;
                                        data[j+1]=data[i+1]-tempi;
                                        data[i] += tempr;
                                        data[i+1] += tempi;
                                }
                                wr = (wtemp=wr)*wpr-wi*wpi+wr;
                                wi=wi*wpr+wtemp*wpi+wi;
                        }
                        mmax=istep;
                }
}
```

```
/*************************************************/
/* Random Number Generator                       */
/*************************************************/
float RAN1(long *idum)
{
        int j;
        long k;
        static long idum2=123456789;
        static long iy=0;
        static long iv[NTAB];
        float temp;

        if(*idum <= 0)
        {
                if(-(*idum) < 1) *idum = 1;
                else *idum = -(*idum);
                idum2=(*idum);
                for(j=NTAB+7;j>=0;j--)
                {
                        k=(*idum)/IQ1;
                        *idum=IA1*(*idum-k*IQ1)-k*IR1;
                        if(*idum<0) *idum += IM1;
                        if(j < NTAB) iv[j] = *idum;
                }
                iy=iv[0];
        }
        k=(*idum)/IQ1;
        *idum=IA1*(*idum-k*IQ1)-k*IR1;
        if(*idum < 0) *idum += IM1;
        k=idum2/IQ2;
        idum2=IA2*(idum2-k*IQ2)-k*IR2;
        if(idum2 < 0) idum2 += IM2;
        j=iy/NDIV;
        iy=iv[j]-idum2;
        iv[j] = *idum;
        if(iy < 1) iy += IMM1;
        if((temp = AM*iy) > RNMX) return RNMX;
        else return temp;
}

float RAN2(long *idum)
{
        int j;
        long k;
        static long idum2=123456789;
        static long iy=0;
```

```
        static long iv[NTAB];
        float temp;

        if(*idum <= 0)
        {
                if(-(*idum) < 1) *idum = 1;
                else *idum = -(*idum);
                idum2=(*idum);
                for(j=NTAB+7;j>=0;j--)
                {
                        k=(*idum)/IQ1;
                        *idum=IA1*(*idum-k*IQ1)-k*IR1;
                        if(*idum<0) *idum += IM1;
                        if(j < NTAB) iv[j] = *idum;
                }
                iy=iv[0];
        }
        k=(*idum)/IQ1;
        *idum=IA1*(*idum-k*IQ1)-k*IR1;
        if(*idum < 0) *idum += IM1;
        k=idum2/IQ2;
        idum2=IA2*(idum2-k*IQ2)-k*IR2;
        if(idum2 < 0) idum2 += IM2;
        j=iy/NDIV;
        iy=iv[j]-idum2;
        iv[j] = *idum;
        if(iy < 1) iy += IMM1;
        if((temp = AM*iy) > RNMX) return RNMX;
        else return temp;
}

float RAN3(long *idum)
{
        int j;
        long k;
        static long idum2=123456789;
        static long iy=0;
        static long iv[NTAB];
        float temp;

        if(*idum <= 0)
        {
                if(-(*idum) < 1) *idum = 1;
                else *idum = -(*idum);
                idum2=(*idum);
                for(j=NTAB+7;j>=0;j--)
```

```
                        {
                                k=(*idum)/IQ1;
                                *idum=IA1*(*idum-k*IQ1)-k*IR1;
                                if(*idum<0) *idum += IM1;
                                if(j < NTAB) iv[j] = *idum;
                        }
                        iy=iv[0];
                }
                k=(*idum)/IQ1;
                *idum=IA1*(*idum-k*IQ1)-k*IR1;
                if(*idum < 0) *idum += IM1;
                k=idum2/IQ2;
                idum2=IA2*(idum2-k*IQ2)-k*IR2;
                if(idum2 < 0) idum2 += IM2;
                j=iy/NDIV;
                iy=iv[j]-idum2;
                iv[j] = *idum;
                if(iy < 1) iy += IMM1;
                if((temp = AM*iy) > RNMX) return RNMX;
                else return temp;
}
/*************************************************/
/* Produces integers between n and m using the     */
/* random number generator output.    */
/*************************************************/
int RAND_INT(float rand, int n, int m)
{
                int rnd_int;
                rnd_int = m + floor(rand * (n - m + 1));
                return rnd_int;
}

/*************************************************/
/* Numerically controlled oscillator for the phase  */
/* revovery unit: sin(*) output.       */
/*************************************************/
double NCO2_SIN(int time_inc)
{
                double fc = 25.0e6, t = 1.0 / 100.0e6;
                double sine;
                sine = (sqrt(2.0) * sin((2.0 * PI * fc * time_inc * t)));
                return sine;
}

/*************************************************/
/* Numerically controlled oscillator for the phase  */
```

```
/* recovery unit: cos(*) output.    */
/*****************************************************/
double NCO2_COS(int time_inc)
{
                double fc = 25.0e6, t = 1.0 / 100.0e6;
                double cosine;
                cosine = (sqrt(2.0) * cos((2.0 * PI * fc * time_inc * t)));
                return cosine;
}
/*****************************************************/
/* Numerically controlled oscillator for the down   */
/* conversion unit: sin(*) output.      */
/*****************************************************/
double NCO1_SIN(int time_inc, double phase_inc)
{
                double fc = 25.0e6, t = 1.0 / 100.0e6;
                double sine;
                sine = (sqrt(2.0) * sin((2.0 * PI * fc * time_inc * t) + phase_inc));
                return sine;
}
/*****************************************************/
/* Numerically controlled oscillator for the down   */
/* conversion unit: cos(*) output.      */
/*****************************************************/
double NCO1_COS(int time_inc, double phase_inc)
{
                double fc = 25.0e6, t = 1.0 / 100.0e6;
                double cosine;
                cosine = (sqrt(2.0) * cos((2.0 * PI * fc * time_inc * t) + phase_inc));
                return cosine;
}


/*****************************************************/
/* Noise Generator                                   */
/*****************************************************/
double NORM_DIST(float rn1, float rn2, float var)
{
                double noise;
                noise = sqrt(-2.0*var*log(rn1))*cos(2*PI*rn2);
                return noise;
}


/*****************************************************/
/* Inegrate and Dump.                                */
```

```
/*************************************************/
void INT_AND_DUMP(double i[], double q[], int sy[], int Ns, int biti[], int bitq[])
{
        int k;
        double tmpi=0.0, tmpq=0.0;
        for(k=1;k<(Ns+1);k++)
        {
                if((k%4) == 0)
                {
                        tmpi += i[k-1];
                        tmpq += q[k-1];
                        if(tmpi > 0.0) biti[k/4] = 1;
                        else biti[k/4] = -1;
                        if(tmpq > 0.0) bitq[k/4] = 1;
                        else bitq[k/4] = -1;
                        if(biti[k/4] == -1)
                        {
                                if(bitq[k/4] == -1)
                                {
                                        sy[k/4] = 2;
                                }
                                else
                                {
                                        sy[k/4] = 3;
                                }
                        }
                        else
                        {
                                if(bitq[k/4] == -1)
                                {
                                        sy[k/4] = 1;
                                }
                                else
                                {
                                        sy[k/4] = 4;
                                }
                        }

                        tmpi = 0.0;
                        tmpq = 0.0;
                }
                else
                {
                        tmpi += i[k-1];
                        tmpq += q[k-1];
                }
```

```
		}
}

int ERR(int s1[], int s2[], int Ns)
{
		int k, er=0;
		for(k=0;k<Ns;k++)
		{
			if(s1[k] != s2[k+1])
			er++;
		}
		return er;
}

double ENERGY(double d[], int Ns)
{
		double sum=0.0, two=2.0;
		int k;
		for(k=0;k<Ns;k++)
		{
			sum += pow(d[k],two);
		}
		sum /= (double) (Ns);
		return sum;
}

/*********************************************/
/* Main Program                                */
/*********************************************/
main()
{
		long idum1=1, idum2=2, idum3=3;
		int i, *symbol, m=1, n=4, j, fft, ifft, CR, TR, UW;
		int err=0;
		int unique_word[15] = {2, 4, 4, 4, 4, 2, 2, 2, 4, 2, 2, 4, 2, 4, 2};
		long N_symbols, N_samples, Lobes=2, K;
		long Samp_per_sym;
		long New_samp_rate, Nsamp;
		float rand;
		double *s, *s_new, *samp_sig, *inph, *quad, *noisy, *tmpy, *symbb;
		int *symb;
		double xx, yy, Ebit, No, Eb_No;
		double Tsamp, Esym, Tsym, fc, phase;
		double Tsamples, energy;
		float noise_var;
		float r1, r2;
```

```
int *bi, *bq;
FILE *inp, *sig, *filspec, *spec, *inpp, *symspec;

inp = fopen("inph.dat","w");
filspec = fopen("filspec.dat","w");
sig = fopen("sig.dat","w");
spec = fopen("spec.dat","w");
inpp = fopen("inpp.dat","w");
symspec = fopen("symspec.dat","w");


yy = 2.0;
xx = 14.0;
ifft = (-1);
fft = 1;
Tsamp=1.0/400.0e6;
Tsym=1.0/25.0e6;
N_symbols = (long) pow(yy,xx);
printf("\n # of symbols is ");
printf("%d\n",N_symbols);
fc = 25.0e6;
Samp_per_sym = (long) (Tsym/Tsamp);
N_samples=N_symbols*Samp_per_sym;
Esym=2.0;
Ebit=Esym/2.0;
CR = 32;
TR = 20;
UW = 15;
phase = 0.0;
New_samp_rate = Samp_per_sym/4;
Nsamp = N_symbols*4;
Eb_No = 10.0;
No = Ebit/Eb_No;
Tsamples=1.0/100.0e6;
noise_var = (No)*2.0;

symbol = (int *) calloc(N_symbols+1,sizeof(int));
symbb = (double *) calloc(2*N_symbols+1,sizeof(double));
s = (double *) calloc(N_samples+1,sizeof(double));
s_new = (double *) calloc(2*N_samples+1,sizeof(double));
inph = (double *) calloc(Nsamp+1,sizeof(double));
quad = (double *) calloc(Nsamp+1,sizeof(double));
symb = (int *) calloc(N_symbols+1,sizeof(int));
samp_sig = (double *) calloc(Nsamp+1,sizeof(double));
noisy = (double *) calloc(Nsamp+1,sizeof(double));
tmpy = (double *) calloc(2*N_samples+1,sizeof(double));
```

```
        bi = (int *) calloc(Nsamp+1,sizeof(int));
        bq = (int *) calloc(Nsamp+1,sizeof(int));

/*****************************************************/
/* Generate the random symbols used to describe the */
/* signal.                                           */
/*****************************************************/
/* phase recovery symbols */
        for(i=0;i<CR;i++)
        {
                symbol[i] = 4;
        }
/* timing recovery symbols */
        for(i=CR;i<(TR+CR);i++)
        {
                if((i%2) == 0) symbol[i] = 4;
                else symbol[i] = 2;
        }
/* unique word symbols */
        for(i=(TR+CR);i<(UW+TR+CR);i++)
        {
                symbol[i] = unique_word[i - (TR+CR)];
        }
        if(N_symbols > 64)
        {
                for(i=(UW+TR+CR);i<N_symbols;i++)
                {
                        rand = RAN1(&idum1);
                        symbol[i] = RAND_INT(rand, n, m);
                }
 }
        for(i=0;i<N_symbols;i++)
        {
                symbb[i] = symbol[i];
        }
        FOUR1(symbb, N_symbols, fft);

        for(i=0;i<2*N_symbols;i++)
        {
                if(symbb[i] < 0.0) symbb[i] *= -1.0;
        }
        for(i=0;i<2*N_symbols;i++)
        {
                fprintf(symspec,"%f\n",symbb[i]);
        }
```

```
/*****************************************************/
/* Generate the signal.                              */
/*****************************************************/
                for(i=0;i<N_symbols;i++)
                {
                        for(j=0;j<Samp_per_sym;j++)
                        {
                                s[i*Samp_per_sym+j] = sqrt(2.0*Esym)
                                *
cos((2.0*PI*fc*((i*Samp_per_sym)+j)*Tsamp)
                                +(((2.0*symbol[i])-1.0)*PI/4.0));
                        }
                }
                printf("\nsignal generated");
                energy = ENERGY(s, N_samples);
                printf("\nThe energy is %f\n",energy);
                for(i=10000;i<(Samp_per_sym*Samp_per_sym+10000);i++)
                {
                        fprintf(sig,"%f\n",s[i]);
                }
/*****************************************************/
/* From this point on, the vector symbol is not      */
/* needed, so free it from memory.                   */
/*****************************************************/
/*              for(i=0;i<N_symbols;i++)
                {
                        fprintf(out1,"%d\n",symbol[i]);
                }
                fclose(out1); */
/*****************************************************/
/* The fft program needs real and imaginary parts    */
/* of each signal sample.  Since the signal is only  */
/* real, the imaginary samples were interleaved in.  */
/*****************************************************/
                for(i=0;i<2*N_samples;i++)
                {
                        if((i % 2) == 0) s_new[i+1] = s[i/2];
                        else s_new[i+1] = 0.0;
                }

/*****************************************************/
/* From this point on, the vector s is not           */
/* needed, so free it from memory.                   */
/*****************************************************/
                free(s);
```

```
/*****************************************************/
/* Take the fft of the signal.                       */
/*****************************************************/
                FOUR1(s_new, N_samples, fft);
                printf("\nspectrum generated");
                for(i=0;i<2*N_samples;i++)
                {
                                if(s_new[i] < 0.0) tmpy[i] = -1.0*s_new[i];
                                else tmpy[i] = s_new[i];
                                fprintf(spec,"%f\n",tmpy[i]);
                }
                fclose(spec);
/*****************************************************/
/* Ideal bandpass filter the signal.   */
/*****************************************************/
                K = (N_samples-2*N_symbols*Lobes)*2;
                for(i=0;i<K;i++)
                {
                                s_new[i+1+2*Lobes*N_symbols] = 0.0;
                }

                printf("\nspectrum filtered");
                for(i=0;i<2*N_samples;i++)
                {
                                if(s_new[i] < 0.0) tmpy[i] = -1.0*s_new[i];
                                else tmpy[i] = s_new[i];
                                fprintf(filspec,"%f\n",tmpy[i]);
                }
                fclose(filspec);

/*****************************************************/
/* Do the ifft to recover the filtered signal.   */
/*****************************************************/
                FOUR1(s_new, N_samples, ifft);
                for(i=0;i<2*N_samples;i += 2)
                {
                                s_new[i+1] /= (double) (N_samples);
                }

/*****************************************************/
/* Sample the filtered signal at 4 samples/symbol.  */
/*****************************************************/

                for(i=0;i<Nsamp;i++)
                {
                                r1 = RAN2(&idum2);
```

```
                        r2 = RAN3(&idum3);
                        samp_sig[i] = s_new[i*2*New_samp_rate+1];
                        noisy[i] = NORM_DIST(r1, r2, noise_var);
                }
                printf("\nsignal sample @ 4s/s");
                energy = ENERGY(samp_sig, Nsamp);
                printf("\nThe sampled sig energy is %f\n",energy);
                energy = ENERGY(noisy, Nsamp);
                printf("\nThe noise energy is %f\n",energy);

                for(i=0;i<Nsamp;i++)
                {
                        samp_sig[i] += noisy[i];
                }
/*****************************************************/
/* From this point on, the vector s_new is not      */
/* needed, so free it from memory.    */
/*****************************************************/
                free(s_new);

                for(i=0;i<Nsamp;i++)
                {
                        inph[i] = samp_sig[i] * NCO1_COS(i,phase);
                        quad[i] = samp_sig[i] * NCO1_SIN(i,phase);
                }
                printf("\ninph and quad data gen");
                for(i=0;i<500;i++)
                {
                        fprintf(inp,"%f\n",inph[i]);
                }
                fclose(inp);
/*****************************************************/
/* From this point on, the vector samp_sig is not   */
/* needed, so free it from memory.    */
/*****************************************************/
                free(samp_sig);
/*****************************************************/
/* Make decision on which symbol was sent using an  */
/* integrate and dump unit.                  */
/*****************************************************/
                INT_AND_DUMP(inph, quad, symb, Nsamp, bi, bq);
/*****************************************************/
/* From this point on, the vectors inph and quad    */
/* are not needed, so free them from memory.    */
/*****************************************************/
                free(inph);
```

```
                free(quad);
                for(i=0;i<Nsamp/32;i++)
                {
                        fprintf(inpp,"%d\n",bi[i]);
                }
                fclose(inpp);
                err = ERR(symbol, symb, N_symbols);
                printf("\n # of errors are %d\n",err);
                printf("\n Eb/No is  %f\n",Eb_No);
}
```

D.1 Demodulator Code

```
#include <stdio.h>
#include <math.h>
/*************************************************************/
/* Pseudo random number generator sub-program.
/* See Fundamentals of Queuing Theory 2nd edition*/
/*Don Gross and Carl Harris.  pg 460-461.
/*************************************************************/
void random(double *r, double *s)
{
double l;
double integer;
double k = 65539.0;
double m = 2147483648.0;

l = k * *r / m;
*r = modf(l, &integer);
*r = *r * m;

*s = *r / (m - 1.0);
}

void add_noise(double *sym_en, double *dB, double *rnd1, double *rnd2,
double *norm)
{
double No, Nout, Nin, noise_BW = 23633000.0, pi = 3.141592654, tmp;
double noise_standard_deviation, mean=0.0, Tsym=1/25000000.0;
double symbol_energy_to_noise_out;
symbol_energy_to_noise_out = pow(10.0 , (*dB / 10.0));
No = *sym_en / symbol_energy_to_noise_out;
noise_standard_deviation = sqrt(No * noise_BW * Tsym);
tmp = -2.0 * log(*rnd1);
*norm = mean + (noise_standard_deviation * sqrt(tmp) * cos(2.0 * pi * *rnd2));
```

```
tmp = tmp + 1;
}

double nco2_sin(int time_inc)
{
double pi = 3.141592654, fc = 25000000.0, t = 1 / 100000000.0;
double sine;
sine = (sqrt(2.0) * sin((2.0 * pi * fc * time_inc * t)));
return sine;
}

double nco2_cos(int time_inc)
{
double pi = 3.141592654, fc = 25000000.0, t = 1 / 100000000.0;
double cosine;
cosine = (sqrt(2.0) * cos((2.0 * pi * fc * time_inc * t)));
return cosine;
}
double nco1_sin(int time_inc, double phase_inc)
{
double pi = 3.141592654, fc = 25000000.0, t = 1 / 100000000.0;
double sine;
sine = (sqrt(2.0) * sin((2.0 * pi * fc * time_inc * t) + phase_inc));
return sine;
}
double nco1_cos(int time_inc, double phase_inc)
{
double pi = 3.141592654, fc = 25000000.0, t = 1 / 100000000.0;
double cosine;
cosine = (sqrt(2.0) * cos((2.0 * pi * fc * time_inc * t) + phase_inc));
return cosine;
}


void low_pass_filter(double in, double *A, double *B, double *C, double *D,
double *E, double *F, double *G, double *H, double *I,
double *J, double *K, double *L, double *M, double *N, double *out)

{
double O, P, Q, R, S, T, U, V, W, X, Y, Z;

*N = *M;
*M = *L;
*L = *K;
*K = *J;
*J = *I;
```

```
*I = *H;
*H = *G;
*G = *F;
*F = *E;
*E = *D;
*D = *C;
*C = *B;
*B = *A;
*A = in;

O = in + *N;
P = -.03125 * O;
Q = *B + *L;
R = .0625 * Q;
S = P + R;
T = *D + *J;
U = -.09375 * T;
V = S + U;
W = *F + *H;
X = .3125 * W;
Y = X + V;

Z = .5 * *G;
*out = (Z + Y);
}

int tru(double ini, double inq, double *Ai, double *Bi, double *Ci,
double *Di, double *Ei, double *Fi, double *Aq, double *Bq,
double *Cq, double *Dq, double *Eq, double *Fq)

{
int outi, outq, out;

*Fi = *Ei;
*Ei = *Di;
*Di = *Ci;
*Ci = *Bi;
*Bi = *Ai;
*Ai = ini;

*Fq = *Eq;
*Eq = *Dq;
*Dq = *Cq;
*Cq = *Bq;
*Bq = *Aq;
*Aq = inq;
```

```
if((((*Ai >= 0.0) && (*Bi >= 0.0) && (*Ci >= 0.0)) &&
((*Di < 0.0) && (*Ei < 0.0) && (*Fi < 0.0))) ||
  (((*Ai < 0.0) && (*Bi < 0.0) && (*Ci < 0.0)) &&
((*Di >= 0.0) && (*Ei >= 0.0) && (*Fi >= 0.0))))
{
outi = 1;
}
else
{
outi = 0;
}

if((((*Aq > 0.0) && (*Bq > 0.0) && (*Cq > 0.0)) &&
((*Dq <= 0.0) && (*Eq <= 0.0) && (*Fq <= 0.0))) ||
  (((*Aq <= 0.0) && (*Bq <= 0.0) && (*Cq <= 0.0)) &&
((*Dq > 0.0) && (*Eq > 0.0) && (*Fq > 0.0))))
{
outq = 1;
}
else
{
outq = 0;
}

if((outi == 1) || (outq == 1))
{
out = 1;
}
else
{
out = 0;
}

return out;
}

int chan_tran(int in, int *AA, int *BB, int *CC, int *DD,
int *EE, int *FF, int *GG, int *HH)
{
int out2;

if((in == 1) && (*DD == 1) && (*HH == 1))
{
out2 = 1;
}
```

```
else
{
out2 = 0;
}

*HH = *GG;
*GG = *FF;
*FF = *EE;
*EE = *DD;
*DD = *CC;
*CC = *BB;
*BB = *AA;
*AA = in;

return out2;
}

void counter(int reset, int *count)
{

if(reset == 1)
{
*count = 0;
}
else
{
*count += 1;
}

if(*count > 3)
{
*count = 0;
}
}

void idu(int set, double data, double *accum, int *hold, double *save_accum,
int *y)
{
int invert;

if(set == 3)
{
*y = 1;
*accum += data;
  *save_accum = *accum;
if(*accum >= 0.0)
```

```
{
invert = 1;
}
else
{
invert = -1;
}
*hold = invert;
  *accum = 0.0;
}
else
{
*y = 0;
*accum += data;
}
}

void integrate(int set1, double data1, double *accum1, double *save_accum1)
{

if(set1 == 3)
{
*accum1 += data1;
*save_accum1 = *accum1;
  *accum1 = 0.0;
}
else
{
*accum1 += data1;
}
}

void pru(int *sample_now, double *i_data, double *q_data, double *save_i_data1,
double *save_q_data1, double *save_i_data2, double *save_q_data2,
double *save_middle_i, double *save_middle_q, int *number,
double *phase_est)
{
double pi = 3.141592654, arg_sample, mag, new_i_data, new_q_data;
double i_average, q_average, phase_est2;
double one, two, three, quant=1.0;

if(*sample_now == 3)
{
*number += 1;

/*******************************************/
```

```
/* Do a rectangular to polar transformation /*
/* on the sample.                          /*
/*****************************************/
if(*i_data == 0.0)
{
arg_sample = pi / 2.0;
}
else
{
arg_sample = (atan2(*q_data,*i_data)) * 4.0;
}

/*****************************************/
/* Do the non-linear transformation on the */
/* magnitude of the sample.                */
/*****************************************/
mag = sqrt((*q_data * *q_data) + (*i_data * *i_data));

/*****************************************/
/* Do a polar to rectangular transformation /*
/* on the sample.                          /*
/*****************************************/
new_i_data = quant * pow(mag,0) * cos((arg_sample + pi));
new_q_data = quant * pow(mag,0) * sin((arg_sample + pi));

/*****************************************/
/* Average the samples over the estimation */
/* period.                                 */
/*****************************************/
if(*number <= 8)
   {
*save_i_data1 += (new_i_data / ((16.0 * 1.0) + 1.0));
*save_q_data1 += (new_q_data / ((16.0 * 1.0) + 1.0));
}
if(*number == 9)
{
*save_middle_i = (new_i_data / ((16.0 * 1.0) + 1.0));
*save_middle_q = (new_q_data / ((16.0 * 1.0) + 1.0));

}
if(*number > 9)
{
   *save_i_data2 += (new_i_data / ((16.0 * 1.0) + 1.0));
*save_q_data2 += (new_q_data / ((16.0 * 1.0) + 1.0));
   }
/*****************************************/
```

```
/* One estimation period is up, take the    */
/* inverse tangent of the averaged data.    */
/*******************************************/
if(*number == 17)
{
i_average = *save_i_data1 + *save_i_data2 + *save_middle_i;
q_average = *save_q_data1 + *save_q_data2 + *save_middle_q;

*phase_est = (atan2(q_average,i_average)) / -4.0;
one = i_average;
if(one < 0.0)
   one *= -1.0;
two = q_average;
if(two < 0.0)
   two *= -1.0;
three = two / one;
if(three < 0.1)
   *phase_est = 0.0;

/*******************************************/
/* Need to use the last averaged data with  */
/* the next averaged data, so save the last */
/* averaged data and start over by finding  */
   /* the new averaged data. */
/*******************************************/
*number = 8;
*save_i_data1 = *save_i_data2;
*save_q_data1 = *save_q_data2;
*save_i_data2 = 0.0;
*save_q_data2 = 0.0;
}
}
}


void main()
{
/*****************************************************************/
/*          Declarations of integers and doubles                 */
/*****************************************************************/
int time, sample, tru_out = 0, chan_tran_out = 0, increment = 0, integ;
int AAA=0, BBB=0, CCC=0, DDD=0, EEE=0, FFF=0, GGG=0, HHH=0, h=0;
int clock = 0, clock1 = 0, clock2 = 0, output_i=0, output_q=0, inc = 0;
int diffi1=0, diffq1=0, tmp3, symbol;

double fake_i=0.0, fake_q=0.0, pru_i=0.0, pru_q=0.0, tmp1, tmp2;
```

```
double phase=0.0, save_ai1=0.0, save_aq1=0.0;
double save_ai2=0.0, save_aq2=0.0, save_m_i=0.0, save_m_q=0.0;
double qpsk_sample, inph, quad, baseband_data_i=0.0, save_ai=0.0,
save_aq=0.0;
double baseband_data_q=0.0, addi=0.0, addq=0.0, inph2, quad2;
double A1=0.0, B1=0.0, C1=0.0, D1=0.0, E1=0.0, F1=0.0, G1=0.0, H1=0.0, I1=0.0;
double J1=0.0, K1=0.0, L1=0.0, M1=0.0, N1=0.0;
double A2=0.0, B2=0.0, C2=0.0, D2=0.0, E2=0.0, F2=0.0, G2=0.0, H2=0.0, I2=0.0;
double J2=0.0, K2=0.0, L2=0.0, M2=0.0, N2=0.0;
double AI=0.0, BI=0.0, CI=0.0, DI=0.0, EI=0.0, FI=0.0;
double AQ=0.0, BQ=0.0, CQ=0.0, DQ=0.0, EQ=0.0, FQ=0.0;
int detected_wordi[12], detected_wordq[12];
int wordi[12] = {-1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, 1};
int wordq[12] = {-1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, 1};
int yes, output_save, int_noise;
double symbol_energy;
double rand1, rand2, seed=1951.0, decibels = 0.0, noise=0.0;
symbol_energy = 0.5;
/*****************************************************************/
/* Files that will be created when running the program. */
/*****************************************************************/
FILE *outi, *outq, *input, *reci, *recq;
FILE *inp, *qua, *nois, *symb;
FILE *ph;
input = fopen("c:\\bk\\demod\\data\\ms.dat","r");
inp = fopen("c:\\bk\\demod\\data\\inph.dat","w");
qua = fopen("c:\\bk\\demod\\data\\quad.dat","w");
outi = fopen("c:\\bk\\demod\\data\\basi.dat","w");
outq = fopen("c:\\bk\\demod\\data\\basq.dat","w");
reci = fopen("c:\\bk\\demod\\data\\recvdi.dat","w");
recq = fopen("c:\\bk\\demod\\data\\recvdq.dat","w");
ph = fopen("c:\\bk\\demod\\data\\phase.dat","w");
nois = fopen("c:\\bk\\demod\\data\\noise.dat","w");
symb = fopen("c:\\bk\\demod\\data\\recsym.dat","w");


for(time = 0; time < 8384; ++time)
{
fscanf(input, "%lf" , &qpsk_sample);
random(&seed, &rand1);
random(&seed, &rand2);
add_noise(&symbol_energy, &decibels, &rand1, &rand2, &noise);
fprintf(nois, "%f\n" , noise);

inph = nco1_cos(time, phase) * qpsk_sample;
  inph += noise;
```

```
inph2 = nco2_cos(time) * qpsk_sample;
   inph2 += noise;

quad = nco1_sin(time, phase) * qpsk_sample;
   quad += noise;
quad2 = nco2_sin(time) * qpsk_sample;
   quad2 += noise;

fprintf(inp,"%f\n",inph);
fprintf(qua,"%f\n",quad);

integrate(increment, inph2, &fake_i, &pru_i);
integrate(increment, quad2, &fake_q, &pru_q);

pru(&increment, &pru_i, &pru_q, &save_ai1, &save_aq1, &save_ai2,
&save_aq2, &save_m_i, &save_m_q, &inc, &phase);

fprintf(ph,"%f\n",phase);

low_pass_filter(inph, &A1, &B1, &C1, &D1, &E1, &F1, &G1, &H1, &I1,
&J1, &K1, &L1, &M1, &N1, &baseband_data_i);

low_pass_filter(quad, &A2, &B2, &C2, &D2, &E2, &F2, &G2, &H2, &I2,
&J2, &K2, &L2, &M2, &N2, &baseband_data_q);

fprintf(outi,"%f\n", baseband_data_i);
fprintf(outq,"%f\n", baseband_data_q);

tru_out = tru(baseband_data_i, baseband_data_q, &AI, &BI, &CI,
&DI, &EI, &FI, &AQ, &BQ, &CQ, &DQ, &EQ, &FQ);

chan_tran_out = chan_tran(tru_out, &AAA, &BBB, &CCC, &DDD,
&EEE, &FFF, &GGG, &HHH);

clock2 = clock1;
clock1 = clock;
counter(chan_tran_out, &clock);

idu(clock2, baseband_data_i, &addi, &output_i, &save_ai, &yes);
idu(clock2, baseband_data_q, &addq, &output_q, &save_aq, &yes);

/*************************************************************/
/* This part of the program is only used to decipher which quadrant    */
/*    the signal is in. */
/*************************************************************/
if((time >= 152) && (time <= 199) && (yes == 1))
```

```
{
detected_wordi[h] = output_i;
detected_wordq[h] = output_q;
if(detected_wordi[h] != wordi[h])
diffi1++;
if(detected_wordq[h] != wordq[h])
diffq1++;
h++;

}

increment += 1;
if(increment > 3)
{
increment = 0;
}

if((time >= 152) && (yes == 1))
{
if((diffi1 > 6) && (diffq1 > 6))
{
if(output_i == -1)
{
output_i = 1;
}
else
{
output_i = -1;
}
if(output_q == -1)
{
output_q = 1;
}
else
{
output_q = -1;
}
}
else
{
if((diffi1 > 6) && (diffq1 <= 6))
{
        output_save = output_q;
if(output_i == -1)
{
output_q = 1;
```

```
}
else
{
output_q = -1;
}
        output_i = output_save;
}
else
{
if((diffi1 <= 6) && (diffq1 > 6))
{
        output_save = output_i;
if(output_q == -1)
{
output_i = 1;
}
else
{
output_i = -1;
}
          output_q = output_save;
}
}
}
}
fprintf(reci,"%d\n", output_i);
fprintf(recq,"%d\n", output_q);
if(yes == 1)
{
if((output_i == -1) && (output_q == -1))
{
symbol = 2;
}
if((output_i == -1) && (output_q == 1))
{
symbol = 3;
}
if((output_i == 1) && (output_q == -1))
{
symbol = 1;
}
if((output_i == 1) && (output_q == 1))
{
symbol = 4;
}
```

```
fprintf(symb, "%d\n",symbol);
}
}

}
```

D.2 Programming The NCO ROM

```
#include <stdio.h>
#include <math.h>

void main()
{
double tmp1, tmp2, tmp3, pi = 3.141592654, store, store2;
int i, tmp, bk;
FILE *out1;
out1 = fopen("nco_rom.dat","w");

for(i = 0; i < 1024; i++)
{
store = cos(i * pi / 512.0);
store2 = sin(i * pi / 512.0);
if(store == 1.0)
{
store = 0.999999;
}
if(store == -1.0)
{
store = -0.99999;
}
tmp1 = store;
if(tmp1 >= 0.0)
{
fprintf(out1,"[0]\n");
}
else
{
fprintf(out1,"[1]\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}

for(bk = 0; bk < 7; bk++)
```

```
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"[%d]\n",tmp);
tmp1 = tmp2;
}

if(store2 == 1.0)
{
store2 = 0.999999;
}
if(store2 == -1.0)
{
store2 = -0.99999;
}
tmp1 = store2;
if(tmp1 >= 0.0)
{
fprintf(out1,"[0]\n");
}
else
{
fprintf(out1,"[1]\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}

for(bk = 0; bk < 7; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"[%d]\n",tmp);
tmp1 = tmp2;
}


}
}
```

D.3 Programming The Non-Linear ROM

```
#include <stdio.h>
```

```
#include <math.h>

void main()
{

double pi = 3.141592654, arg_sample, mag, new_i_data, new_q_data;
double i_average, q_average;
double one, two, three, quant=1.0, i_data, q_data, phase_est;
double tmp1, tmp2, tmp3;
int sample_now = 3, a=0, tmp, bk;
FILE *out1;
out1 = fopen("vnli.dat","w");

for(i_data = 0; i_data < 32; i_data++)
{
for(q_data = 0; q_data < 32; q_data++)
{
 /*******************************************/
/* Do a rectangular to polar transformation /*
  /* on the samples.                        /*
/*******************************************/
if(i_data == 0)
{
if(q_data == 0)
{
arg_sample = 0.0;
}
else
{
arg_sample = 4.0 * pi / 2.0;
}
}
else
{
arg_sample = (atan2(q_data,i_data)) * 4.0;
     }

/*******************************************/
/* Do the non-linear transformation on the  */
  /* magnitude of the sample.               */
/*******************************************/
mag = sqrt((q_data * q_data) + (i_data * i_data));

  /*******************************************/
/* Do a polar to rectangular transformation /*
  /* on the sample.                         /*
```

```
/*****************************************/
new_i_data = quant * pow(mag,a) * cos((arg_sample + pi));
new_q_data = quant * pow(mag,a) * sin((arg_sample + pi));
if(new_i_data == 1.0)
new_i_data = 0.999999;
if(new_i_data == -1.00)
new_i_data = -0.999999;
if(new_q_data == 1.0)
new_q_data = 0.999999;
if(new_q_data == -1.00)
new_q_data = -0.999999;

tmp1 = new_i_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}

tmp1 = new_q_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
```

```
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}



}

for(q_data = -32; q_data < 0; q_data++)
{
/*****************************************/
/* Do a rectangular to polar transformation /*
  /* on the sample.                        /*
/*****************************************/
if(i_data == 0)
{
      arg_sample = 4.0 * 3.0 * pi / 2.0;
}
else
{
arg_sample = (atan2(q_data,i_data)) * 4.0;
    }

/*****************************************/
/* Do the non-linear transformation on the  */
  /* magnitude of the sample.              */
/*****************************************/
mag = sqrt((q_data * q_data) + (i_data * i_data));

  /*****************************************/
/* Do a polar to rectangular transformation /*
  /* on the sample.                        /*
/*****************************************/
new_i_data = quant * pow(mag,a) * cos((arg_sample + pi));
new_q_data = quant * pow(mag,a) * sin((arg_sample + pi));
if(new_i_data == 1.0)
new_i_data = 0.999999;
if(new_i_data == -1.00)
new_i_data = -0.999999;
if(new_q_data == 1.0)
new_q_data = 0.999999;
if(new_q_data == -1.00)
new_q_data = -0.999999;
tmp1 = new_i_data;
```

```
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;

}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}

tmp1 = new_q_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}

}
}
  for(i_data = -32; i_data < 0; i_data++)
{
for(q_data = 0; q_data < 32; q_data++)
{
```

```
/*******************************************/
/* Do a rectangular to polar transformation /*
 /* on the sample.                          /*
/*******************************************/

arg_sample = (atan2(q_data,i_data)) * 4.0;

/*******************************************/
/* Do the non-linear transformation on the  */
 /* magnitude of the sample.                */
/*******************************************/
mag = sqrt((q_data * q_data) + (i_data * i_data));

 /*******************************************/
/* Do a polar to rectangular transformation /*
 /* on the sample.                          /*
/*******************************************/
new_i_data = quant * pow(mag,a) * cos((arg_sample + pi));
new_q_data = quant * pow(mag,a) * sin((arg_sample + pi));
if(new_i_data == 1.0)
new_i_data = 0.999999;
if(new_i_data == -1.00)
new_i_data = -0.999999;
if(new_q_data == 1.0)
new_q_data = 0.999999;
if(new_q_data == -1.00)
new_q_data = -0.999999;
tmp1 = new_i_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}
```

```
tmp1 = new_q_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;
}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}



}
for(q_data = -32; q_data < 0; q_data++)
{
/*****************************************/
/* Do a rectangular to polar transformation /*
  /* on the sample.                        /*
/*****************************************/

arg_sample = (atan2(q_data,i_data)) * 4.0;

/*****************************************/
/* Do the non-linear transformation on the  */
  /* magnitude of the sample.              */
/*****************************************/
mag = sqrt((q_data * q_data) + (i_data * i_data));

  /*****************************************/
/* Do a polar to rectangular transformation /*
  /* on the sample.                        /*
/*****************************************/
new_i_data = quant * pow(mag,a) * cos((arg_sample + pi));
new_q_data = quant * pow(mag,a) * sin((arg_sample + pi));
if(new_i_data == 1.0)
```

```
new_i_data = 0.999999;
if(new_i_data == -1.00)
new_i_data = -0.999999;
if(new_q_data == 1.0)
new_q_data = 0.999999;
if(new_q_data == -1.00)
new_q_data = -0.999999;
tmp1 = new_i_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;

}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
tmp1 = tmp2;
}

tmp1 = new_q_data;
if(tmp1 >= 0)
{
fprintf(out1,"0\n");
}
else
{
fprintf(out1,"1\n");
tmp1 *= -1.0;
tmp1 = 1.0 - tmp1;

}
for(bk = 0; bk < 5; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
fprintf(out1,"%d\n",tmp);
```

```
tmp1 = tmp2;
}


}
}
}
```

D.4 Programming The Phase Estimate ROM

```
#include <stdio.h>
#include <math.h>

void main()
{

double one, two, three, i_data, q_data, phase_est;
double tmp1, tmp2, tmp3, pi = 3.141592654, store;
int bk, tmp;
FILE *out1;
out1 = fopen("phase.dat","w");

for(i_data = 0.0; i_data < 8.0; i_data++)
{
for(q_data = 0.0; q_data < 8.0; q_data++)
{
if(i_data == 0.0)
{
if(q_data == 0.0)
{
phase_est = 0.0;
}
else
{
phase_est = pi / 8.0;
}
}
else
{
phase_est = (atan2(q_data,i_data)) / -4.0;
if(phase_est < 0.0)
{
phase_est += (2.0 * pi);
}
```

```
}
store = phase_est * 512.0 / (pi * 1024.0);
if(store == 1.0)
{
store = 0.999999;
}
tmp1 = store;
for(bk = 0.0; bk < 10.0; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
if(bk >= 2.0)
{
fprintf(out1,"[%d]\n",tmp);
}
tmp1 = tmp2;
}
}
for(q_data = -8.0; q_data < 0.0; q_data++)
{
if(i_data == 0.0)
{
phase_est = -1.0 * pi / 8.0;
}
else
{
phase_est = (atan2(q_data,i_data)) / -4.0;
if(phase_est < 0.0)
{
phase_est += (2.0 * pi);
}
}
store = phase_est * 512.0 / (pi * 1024.0);
if(store == 1.0)
{
store = 0.999999;
}
tmp1 = store;
for(bk = 0.0; bk < 10.0; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
if(bk >= 2.0)
{
```

```
fprintf(out1,"[%d]\n",tmp);
}


tmp1 = tmp2;
}
}
}
for(i_data = -8.0; i_data < 0.0; i_data++)
{
for(q_data = 0.0; q_data < 8.0; q_data++)
   {
phase_est = (atan2(q_data,i_data)) / -4.0;
if(phase_est < 0.0)
{
phase_est += (2.0 * pi);
}

store = phase_est * 512.0 / (pi * 1024.0);
if(store == 1.0)
{
store = 0.999999;
}
tmp1 = store;
for(bk = 0.0; bk < 10.0; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
if(bk >= 2.0)
{
fprintf(out1,"[%d]\n",tmp);
}


tmp1 = tmp2;
}
}
for(q_data = -8.0; q_data < 0.0; q_data++)
{
phase_est = (atan2(q_data,i_data)) / -4.0;
if(phase_est < 0.0)
{
phase_est += (2.0 * pi);
}
```

```
store = phase_est * 512.0 / (pi * 1024.0);
if(store == 1.0)
{
store = 0.999999;
}
tmp1 = store;
for(bk = 0.0; bk < 10.0; bk++)
{
tmp1 *= 2.0;
tmp2 = modf(tmp1, &tmp3);
tmp = 1 * tmp3;
if(bk >= 2.0)
{
fprintf(out1,"[%d]\n",tmp);
}


tmp1 = tmp2;
}
}
}
}
```

References

1.) M. Miller, B. Vucetic, L. Berry, *Satellite Communications: Mobile and Fixed Services*, Klewer Academic Publishers, Norwell, Massachusetts, 1993, page 14.

2.) David Wagner, "VLSI Architecture Design of TDM High Data Rate QPSK Demodulator", University of Toledo, September 23, 1992.

3.) Simon Haykin, *Digital Communications*, John Wiley and Sons, New York, New York,1988, pages 284-290.

4.) Simon Haykin, *An Introduction to Analog and Digital Communications*, John Wiley and Sons, New York, New York,1989, page 566.

5.) Charles Baugh, Bruce Wooley, "A Two's Complement Parallel Array Multiplication Algorithm", IEEE Transactions on Computers, Vol. C-22, No. 12, December 1973.

6.) Fang Lu, Henry Samueli, "A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design", IEEE Journal of Solid-State Circuits, Vol. No. 2, February 1993.

7.) Quangfu Zhao, Yoshiaki Tadokoro, "A Simple Design of FIR Filters with Powers of Two Coefficients", IEEE Transactions on Circuits and Systems, Vol. 35, No. 5, May 1988.

8.) Yong Lim, Sydney Parker, "FIR Filter Design Over a Discrete Powers of Two Coefficient Space", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-31, No. 3, June 1983.

9.) A.J. Viterbi and A.M. Viterbi, "Nonlinear Estimation of PSK Modulated Carrier Phase With Application To Burst Digital Transmission", IEEE Transactions on Information Theory, pages 543-51, July 1983.

10.) T. Pratt and C. Bostian, *Satellite Communications*, Wiley and Sons, New York, New York, 1986, pages 245-47.

11.) Neil Weste, Kamran Eshraghian, *Principles of CMOS VLSI Design, A System Perspective, 2nd Edition*, Addison-Wesley Publishing Co., New York, New York, 1993, Chapter 4-5.

12.) Yong Dhong, C.P. Tsang, "High Speed CMOS POS PLA Using Pre-discharged OR Array and Charge Sharing AND Array", IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, Vol. 39, No. 8, August 1992

13.) William H. Press, et. al., *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1988, page 288.

14.) M. Jeruchim, P. Balaban, K. Shanmugan, *Simulation of Communication Systems*, Plenum Press, New York, New York, 1992, pages 1-11.

15.) Kurt Mhueller, Maarkus Muller, "Timing Recovery in Digital Synchronous Data Receivers", IEEE Transactions on Communications, Vol. COM-24, No. 5. May 1974.

16.) Floyd Gardner, "A BPSK/QPSK Timing-Error Detector for Sampled Receivers", IEEE Transactions on Communications, Vol. COM-34, No. 5, May 1986.

17.) Benette Wong, Henry Samueli, "A 200 MHz All Digital QAM Modulator and Demodulator in 1.2m CMOS for Digital Radio Applications". IEEE Journal of Solid-State Circuits, Vol. 26, No. 12, December 1991.